BRITTLE FRACTURE MODELING WITH A SURFACE TENSION EXCESS PROPERTY

A Dissertation

by

LAUREN ANN FERGUSON

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Jay R. Walton
Committee Members,	Raytcho Lazarov
	Francis J. Narcowich
	Vikram Kinra
Department Head,	Emil Straube

December 2012

Major Subject: Mathematics

ABSTRACT

The classical theory of linear elastic fracture mechanics for a quasi-static crack in an infinite linear elastic body has two significant mathematical inconsistencies: it predicts unbounded crack-tip stresses and an elliptical crack opening profile. A new theory of fracture developed by Sendova and Walton, based on extending continuum mechanics to the nanoscale, corrects these erroneous effects. The fundamental attribute of this theory is the use of a dividing surface to describe the material interface. The dividing surface is endowed with an excess property, namely surface tension, which accounts for atomistic effects in the interfacial region. When the surface tension is taken to be a constant, Sendova and Walton show that the theory reduces the crack-tip stress from a square root to a logarithmic singularity and yields a finite angle opening profile. In addition, they show that if the surface tension depends on curvature, the theory completely removes the stress singularity at the crack-tip, for all but countably many values of the two surface tension parameters, and yields a cusp-like opening profile.

In this work, we develop a numerical model using the finite element method for the Sendova-Walton fracture theory applied to the classical Griffith crack problem in the case of constant surface tension. We show that the numerical model behaves as predicted by the theory, yielding a reduced crack-tip singularity and a finite opening angle for all nonzero values of the constant surface tension. We also lay the groundwork for the numerical implementation of the curvature-dependent model by constructing an algorithm to determine the appropriate threshold values for the surface tension parameters that guarantee bounded crack-tip stresses. These values can then be directly applied to the forthcoming numerical model.

To my crackerjack family $\sim$ Waylon, Carol, and Bethany

ACKNOWLEDGEMENTS

I am glad of the chance to thank some very instrumental people who made the completion of this dissertation possible.

First, I am so grateful to my advisor, Jay Walton, who was always available to listen to problems I encountered in my research and never failed to provide new insights and inspiration to help me start sorting them out. He has been a constant source of support, even from my first visit to the Mathematics Department. I am so glad he gave me the opportunity first to participate in the NSF IGERT program and then to work with him as his student.

I also want to thank the other members of my committee: Raytcho Lazarov, Francis Narcowich, and Vikram Kinra, who all took the time to help me when I had questions, both when taking classes from them or when grappling with research. I am also grateful to Andrea Bonito for substituting at my defense and giving me some very useful suggestions on the dissertation.

I am especially indebted to Sebastian Pauletti, Wolfgang Bangerth, and Guido Kanschat for helping me with a multitude of questions about `deal.II`. I am extremely appreciative of their help and patience in working through coding issues with me.

I would like to thank Tsvetanka Sendova for her timely advice and collaboration. The groundwork she provided in her own dissertation research made mine possible.

Of course, I would have been completely overwhelmed by stress if it hadn't been for the constant encouragement, support, and prayers from my family and friends. Thanks especially to my parents, Waylon and Carol Ferguson, and my sister, Bethany, who patiently listened to me describe my research woes in meticulous detail. They may

not have understood most of it, but they listened well, asked good questions, and kept believing in me, often giving me new ideas in the process.

My friends at school heard these research trials, and occasional "Eureka!" moments, with greater understanding and often suggested new avenues to try. They also provided much-needed breaks! Thanks especially to Maya, Tracy, Ruifang, Jeanette, Aditi, and Xuebing.

Finally, all thanks to my Lord Jesus for his many blessings and unfailing love.

NOMENCLATURE

DMB	Differential Momentum Balance
DOF	Degrees of Freedom
FE	Finite Element
FEM	Finite Element Method
JMB	Jump Momentum Balance
LEFM	Linear Elastic Fracture Mechanics
ST	Surface Tension

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

LIST OF LISTINGS

LISTING Page

1. INTRODUCTION

Catastrophic failure seems to be inevitable in mechanical systems, from malfunction of medical implants, such as artificial heart valves and pacemakers, to total collapse of large structures like bridges. Often these failures can be attributed to the fracture of a particular component, making it imperative to know when to repair or replace these components before fracture becomes imminent. To predict the usable lifetime of a component, we rely on numerical simulations that model how material bodies respond under various conditions. In particular, we can model how cracks in a material body can form and grow and eventually cause a fracture of the body into separate pieces. In this dissertation, we present a new numerical model of brittle fracture for a cracked material body which accurately predicts the stress field not only in the bulk of the body, but also near the fracture surfaces, a significant improvement over conventional models.

Fracture of brittle materials has been widely studied during the last century, largely motivated by the sudden fracturing of structures whose designs followed accepted specifications. Examples include brittle hull fractures of ships and failure of pressurized airplane cabins (Erdogan 2000). It was known that the machining process introduced small ($\sim$micron) crack-like defects on the surface of a component, but it was soon discovered that under certain loadings, geometries, temperatures, etc., a microscale defect could grow into a macroscale crack, eventually propagating across the material until total fracture occurred. It was clear that treating bodies as perfect materials, ignoring the presence of small defects (micro-cracks), was not representative of the true physical situation.

Early models of cracked material bodies used a continuum mechanics framework, which was not only effective, but involved straightforward mathematics. (A comprehensive discussion of these early models can be found in (Broberg 1999).) In his pioneering work, Griffith (1921) takes the view that crack growth is governed by the competition between the elastic energy stored in the body and the surface energy of the crack. He rejected the previously held notion that a material's strength or fracture criterion should be based on the maximum stress in the entire body, instead showing such a criterion should be energy based. By balancing energy, he computed the fracture stress as a function of crack size to give a better fracture criterion (see Erdogan 2000).

Although useful fracture criteria have emerged since then (in particular Irwin's stress intensity factor and fracture toughness criterion (Irwin 1948)), the major drawback of the continuum mechanics framework is precisely its strict continuum assumption which oversimplifies the physical situation by excluding any interatomic physics. The archetypal continuum model of fracture is linear elastic fracture mechanics (LEFM), which is based on the rupture theory of Griffith (1921). It considers only material bodies that are linearly elastic and assumes that all strains are small. The LEFM model is excellent at predicting the stress fields in the bulk material away from the crack, but it is highly inaccurate near the crack surfaces. In particular, it predicts infinite stress (and a corresponding infinite strain) at the crack tips, which is physically impossible and violates the LEFM small strain assumption. It also predicts an elliptical crack opening profile that is inconsistent with experimental data, which shows that the crack tips have a cusp-like opening profile. Even more erroneous for an interface fracture between two solids, LEFM predicts interpenetration of the fracture surfaces.

These inconsistencies in the LEFM model are a direct consequence of modeling the mechanical behavior of a material solely at the continuum scale, without explicitly considering effects that occur at the atomistic scale. One way to approach continuum-scale modeling is to view it as an averaging of discrete atomistic-scale models in which atoms are represented as small balls over which interatomic forces act with an effective length scale equal to the ball radius. These continuum-scale models are effective in the bulk, yielding good approximations to the bulk stress-strain relations. However, they give very poor approximations to these relations near material interfaces where atomic-scale interactions between atoms of two distinct material phases complicate the mechanical response of the material. Accounting for these atomistic effects would improve the accuracy of the model and allow for the prediction of crack growth and fracture. In particular, this correction might allow for a maximum stress fracture criterion. Such an improvement in the prediction of the stress field in the body is especially critical for modeling devices that operate on small scales, which continue to shrink as today's technological advances grow. However, while there is universal consensus that atomistic effects should be included in any fracture model, there is much debate over exactly how these effects should be incorporated.

One idea is to scrap the continuum approach and model all atomic interactions in the body. Typically, these methods have been used to predict the speed and direction of propagating cracks. Computational molecular dynamics is one such atomistic approach. Abraham (2001) (see also Abraham, Brodbeck, Rudge, and Xu 1997) has used this method to conduct simulations of brittle fracture. His main focus was on predicting the speed of crack propagation and he showed that his atomistic calculations agreed more consistently with experimental data than the classical theory. However, even for simple two-dimensional bodies that are only a few atoms thick, this method can involve upwards of one billion atoms and requires a dedicated supercomputer to

run the simulation. Although we continue to see tremendous advances in technology and data storage, this is a serious limitation of this approach. In addition, atomistic methods tend to be highly dependent on material properties, geometries, and loading conditions.

Another downside to purely atomistic models is that they are not as accurate at predicting stress fields in the bulk material away from the crack when compared with continuum models. This has spurred an effort to create mixed atomistic and continuum models to exploit the advantages of both approaches. This is a difficult task given that such a hybrid must span length scales to be accurate both at the nanoscale region surrounding the crack surfaces and the macroscale region in which the bulk resides.

There are two main schools of thought when it comes to coupling atomistic and continuum methods: concurrent and hierarchical coupling (Belytschko and Xiao 2004). The first is an explicit coupling in which the body is decomposed into two regions, one atomistic and one continuum. Typically, the continuum is modeled using finite elements (FEs). In the atomistic region, atomic energies are computed using potentials, generally a Lennard-Jones potential, an embedded-atom method, or a Stillinger-Weber type framework (Curtin and Miller 2003; Belytschko and Xiao 2004). An important example is the quasicontinuum method of Tadmor, Phillips, and Ortiz (1996). Their method employs standard continuum mechanics formulations to describe the boundary value problem, but explicitly handles atomic interactions through the computation of strain energy at the quadrature points. This enables a reduction in the degrees of freedom required by atomistic methods while retaining symmetry information of the underlying atomic lattice (see also Miller, Tadmor, Phillips, and Ortiz 1998; Miller and Tadmor 2007).

The main drawback of the quasicontinuum and other concurrent coupling methods (see Curtin and Miller 2003) is that the simple interatomic potentials and force laws they use to approximate the non-local atomic interactions can be highly inaccurate, especially for large strains. Even locally, many-body potentials only give approximations to the true interatomic forces and are determined by fitting specific parameters to yield results comparable with experimental data. Abraham (2001) admits that these insufficiently accurate interatomic force approximations open himself up to the charge that this approach prohibits an examination of "real" materials, but he justifies his choice by stating his desire to study only "'generic' features ... common to a large class of real physical systems". The application-specific parameters required to construct the potentials also make it difficult to adapt atomistic methods for general use. In addition, the incompatibility of the non-local calculations for the atomistic region and the local computations in the FE region produce non-physical effects (so-called "ghost forces") that must be corrected (Curtin and Miller 2003; Belytschko and Xiao 2004).

In contrast, hierarchical coupling is an implicit coupling method in which information gleaned from the atomistic scale is passed to the continuum model in the form of effective properties or constitutive laws (Curtin and Miller 2003). This allows one to incorporate the effects of atomistic phenomena without having to model any of the actual atomic interactions. For example, Dal Maso, Francfort, and Toader (2005) have developed a variational model of quasi-static brittle crack growth, based directly on Griffith's theory, in which growth is modeled by an evolution of minimum energy configurations. They substitute Griffith's surface energy, which lacks cohesive force effects, with a Barenblatt-type energy that accounts for these forces. They also replace the stability condition arising from the variational formulation with a local or global minimality principle (Bourdin, Francfort, and Marigo 2008). This provides a critical

yield stress criterion for crack initiation.

However, these variational methods (see also Francfort and Larsen 2003; Francfort and Garroni 2006) have typically been applied to problems with simple geometries and loadings in one or two dimensions and may not be easily adapted to more complex settings. In fact, Burke, Ortner, and Süli (2010) observe that while the variational model of Francfort and Marigo (1998) can be successfully used for predicting the path of an evolving crack, "In other respects, it may fall short of physical reality even on a qualitative level." They also note that their own variational model algorithms can be sensitive to the parameters used.

Even so, hierarchical coupling can still be used to answer the crucial question in fracture modeling concerning bridging length scales: how to model the interfacial region immediately around the fracture surface. The interfacial physics required to describe a material interface is complex and not yet well-understood, but its necessity for describing various material processes in addition to fracture (e.g., friction, wetting, corrosion, etc.) has garnered it a great deal of recent research. In particular, Slattery, Oh, and Fu (2004) describe a hierarchical coupling method for modeling material interfaces that incorporates the necessary interfacial physics into a fully continuum framework through an extension of continuum mechanics to the nanoscale. This is accomplished by assuming the existence of a two-dimensional *dividing surface*, which is a geometric description of a hypothetical surface lying along the material surface at the interface.

This dividing surface, which was first proposed by Gibbs (1928), is used to approximate the complex mechanical behavior that occurs in the neighborhood of the material interface. It accounts for atomistic effects through an ascribed surface excess property (e.g., internal energy, free energy, stress, mass, temperature, etc.). Slattery et al. (2004) tested their model on three different applications, including predicting

static contact angles and surface tension of the n-alkanes, and showed their method agrees well with experimental data and is superior to the computational predictions of previous methods.

This new material interface model was applied in the context of fracture by Oh, Walton, and Slattery (2006). They use the method described above to characterize the interfacial region around a crack surface. They consider an opened crack so that the dividing surface separates a material phase on one side from a gas phase interior to the crack (usually a vacuum) on the other. The interfacial region is the immediate neighborhood of approximately 10nm around the crack surface. The effects of long-range intermolecular forces from adjacent phases are accounted for in two ways. First, as mentioned above, they assign an excess property to the dividing surface. The main properties they consider are surface tension and surface energy. Second, they add a correction term to the bulk description of material behavior along the interface. They applied their fracture theory to the classical Griffith crack problem of a mode-I crack in an infinite linear elastic body loaded under tensile stress. They used singular perturbation methods to solve the problem and predict both the shape of the opened crack surfaces and the stress distribution in the body, particularly in the interfacial region. They predicted a finite slope at the crack tip and claimed that the predicted stress at the crack tip was finite.

The significant advantage to the approach of Oh et al. (2006) is that it is independent of any specific application or implementation, requiring no adjustable parameters. They conjecture that both the surface tension (or energy) excess property and the bulk correction term are necessary to remove the crack-tip stress singularity that appears in the LEFM model. However, they do not explicitly prove this supposition. In fact, they a priori assume the existence of an appropriate constant surface tension (or energy)

that results in finite crack-tip stresses. This assumption implicitly asserts an a priori assumption on the stress-deformation behavior.

In contrast, we contend that the stress-deformation behavior must be modeled constitutively. Consequently, we consider the fracture theory of Sendova and Walton (2010) which is analogous to that of Oh et al. (2006) and assigns surface tension as the excess property of the dividing surface, but makes no a priori assumption on the values of stress and strain at the crack tip. They consider four different models for a mode-I crack, using either a constant or curvature-dependent surface tension and either a zero or nonzero mutual body force. In the case of zero body force, they show that the constant surface tension model does not remove the stress singularity at the crack tip, but does reduce it from a square root singularity to a logarithmic singularity. This model also yields a finite crack-tip opening angle instead of the elliptical opening predicted by LEFM. (Note that by finite opening angle, we mean an acute angle with finite slope.) For the model using curvature-dependent surface tension, they prove that the stress singularity at the crack tip is actually removed and the crack surfaces at the tip form a cusp. Thus they prove that curvature-dependent surface tension is sufficient to correct the inconsistencies of the LEFM model for a mode-I crack, refuting the previous conjecture of Oh et al. (2006) that a nonzero body force correction term is also necessary. Since this model predicts finite stresses at the crack tips, it also makes it possible to determine a fracture criterion based on critical crack-tip stress.

The vast improvement that the Sendova-Walton fracture theory shows over the LEFM model and other fracture theories that have attempted to bridge the continuum-to-atomistic length scales makes it highly desirable to use in a variety of fracture modeling applications. Towards this end, we have developed a numerical model of brittle fracture based on the Sendova-Walton fracture theory for the case of constant surface tension. We applied the theory to the classical Griffith crack problem and used

the finite element method (FEM) for its implementation. Our model yields an accurate prediction of both the total displacement of the body due to the applied loading, including the opening profile of the crack, and the stress distribution throughout the body. Additionally, we lay the groundwork for the development of the model in the case of curvature-dependent surface tension by constructing an algorithm for computing threshold values of the surface tension parameters that guarantee bounded crack-tip stresses.

The rest of this work is organized as follows: In Section 2, we discuss in more detail the mathematical theory of brittle fracture developed by Sendova and Walton (2010) and its application to our problem. We also formulate the variational problem with constant surface tension that we will solve using FEM and show that it is well-posed.

We elaborate on the numerical model we created to solve this problem and present the results of this implementation in Section 3. In particular, we will show that this constant surface tension model agrees with the theory in that it predicts a finite opening angle at the crack tip and logarithmically singular crack-tip stresses. Since there are no adjustable parameters, this model can be easily adapted to more complex scenarios, including problems with different types of materials, bodies with various geometries and loadings, and the inclusion of time dependency for propagating cracks.

Although Sendova and Walton show the existence of surface tension parameters that yield bounded stresses in the case of curvature-dependent surface tension, they do not address what these parameters should be. We present a new existence proof in Section 4 that more easily lends itself to their determination and use it to construct an algorithm for generating threshold values for these parameters that guarantee bounded crack-tip stresses.

Finally, we give some concluding remarks and ideas for future work in Section 5. We are particularly interested in developing the numerical model for the curvature-

dependent case and give a brief overview of the challenges involved in its implementation.

2. FRACTURE MODEL FOR THE GRIFFITH CRACK PROBLEM

We consider the numerical simulation of the classical mode-I Griffith crack problem in which an infinite linear elastic body containing a straight, transverse crack is subjected to uniform, far-field tensile loading (σ). In this quasi-static problem, we predict how the crack will deform under the loading, but we assume that no crack growth due to the breaking of atomic bonds occurs. The numerical model we propose is an application of the finite element method (FEM) to the mathematical fracture theory developed by Sendova and Walton (2010).

For the scope of this dissertation, we consider only the two-dimensional case, but note that the model is easily extended to three dimensions. Consistent with Sendova and Walton, we also nondimensionalize the system both by Young's modulus (E) and by crack half-length (a). The infinite plane, denoted Ω, subjected to the loading σ and containing the nondimensionalized crack surface (Σ) is shown below in Figure 2.1. We assume that the sides are traction-free.

2.1 The Sendova-Walton Model

To determine the governing equations for this problem, Sendova and Walton balance the forces acting on an arbitrary part of the body that intersects the dividing surface, which coincides with the crack surface Σ. This is done in the reference configuration, in which terms are subscripted by $(_\kappa)$. Note that Σ is the union of the upper and lower crack surfaces, which in the reference configuration are parametrized by

$$\Sigma^{\pm} = \{\mathbf{X} : |X_1| \leq 1,\ X_2 = 0^{\pm}\}. \tag{2.1}$$

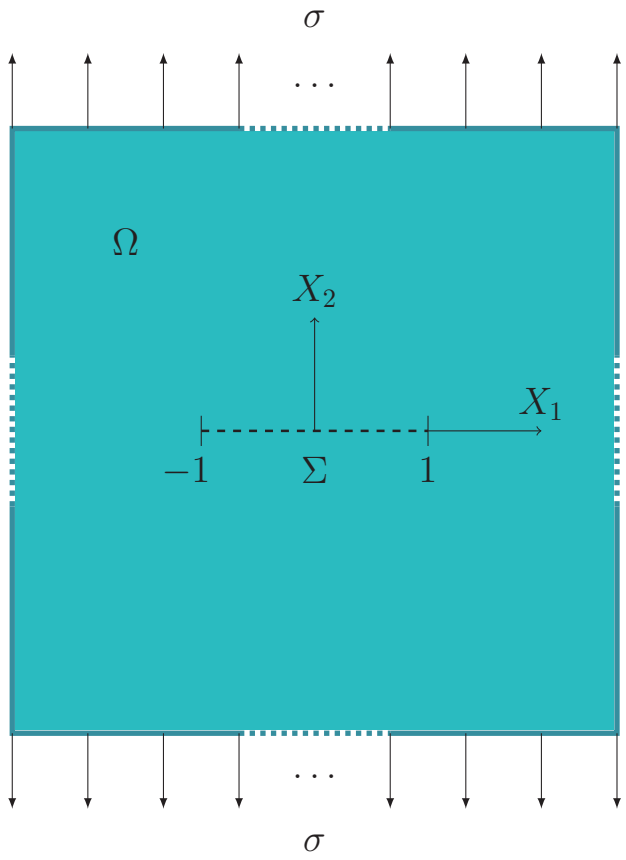


Figure 2.1 Infinite plane Ω with included crack under uniform tensile loading

The force balance results in two balance equations: a differential momentum balance on the body and a jump momentum balance across both the upper and lower crack surfaces.

The equation for the differential momentum balance (DMB) on Ω turns out to be identical to its counterpart derived from the traditional linear elastic fracture mechanics (LEFM) model. In particular, we have

$$\mathrm{Div}\mathbf{T}_\kappa + \mathbf{b}_\kappa = 0, \quad \text{in} \ \ \Omega, \tag{2.2}$$

where $\mathbf{T}_\kappa$ is the first Piola-Kirchhoff stress and $\mathbf{b}_\kappa$ is a mutual body force used as the bulk correction term. However, ascribing surface tension to the dividing surface yields a new expression for the jump momentum balance (JMB) along the crack surface, which is given by

$$J(\mathrm{div}_{(\sigma)}\,\mathbf{T}^{(\sigma)}\otimes\mathbf{n}^{\mp})_m\mathbf{F}^{-T}\mathbf{N}^{\mp}+[\![\mathbf{T}_\kappa]\!]\,\mathbf{N}^{\mp}=0,\ \text{ on }\ \Sigma^{\pm}, \tag{2.3}$$

where $\mathbf{F}$ is the deformation gradient, $J=\det\mathbf{F}$, $\mathrm{div}_{(\sigma)}$ indicates surface divergence, and $\mathbf{N}^-[\mathbf{N}^+]$ and $\mathbf{n}^-[\mathbf{n}^+]$ are outward unit normals to the upper[lower] crack profile in the reference and deformed configuration, respectively, pointing into the bulk. The subscript $(_m)$ refers to the material description of the quantity in parentheses. The double bracket indicates the jump across the fracture surface from the material phase of the body to the phase bounded by the opened fracture surfaces, i.e., $[[\mathbf{T}_\kappa]]=\mathbf{T}_\kappa^{(bulk)}-\mathbf{T}_\kappa^{(C)}$. We assume that the phase bounded by the opened fracture surfaces is a vacuum, i.e., $\mathbf{T}_\kappa^{(C)}=0$. In this case, $[[\mathbf{T}_\kappa]]=\mathbf{T}_\kappa^{(bulk)}$ and henceforth we drop the $(^{(bulk)})$ superscript. For the two-dimensional problem described above, we have the component form of the stress tensor

$$\mathbf{T}_\kappa=\begin{pmatrix}\sigma_{11}&\sigma_{12}\\ \sigma_{21}&\sigma_{22}\end{pmatrix}. \tag{2.4}$$

In order to balance forces, we assumed the existence of a surface Cauchy stress tensor $\mathbf{T}^{(\sigma)}$ in accordance with Cauchy's stress theorem (Gurtin 2003, p. 101), or rather with the analogous theorem for surface stress given by Slattery, Sagis, and Oh (2007, p. 114). It is shown in (Slattery, Oh, and Fu 2004) that the surface Cauchy stress tensor may be written as

$$\mathbf{T}^{(\sigma)}=\tilde{\gamma}\mathbf{P}, \tag{2.5}$$

where $\tilde{\gamma}$ is surface tension and $\mathbf{P}$ is a projection tensor.

The outward unit normal $\mathbf{n}^-$ in the deformed configuration is computed from the shape of the crack surface, which is determined by the displacement vector $\mathbf{u}_\kappa = \langle u_1, u_2 \rangle^\mathsf{T}$. We denote the first derivatives of displacement by $u_{i,j} = \frac{\partial u_i}{\partial X_j}$, with analogous notation for higher derivatives. The surface tension is a function only of the tangential component, i.e., $\tilde{\gamma} = \tilde{\gamma}(X_1)$. With this notation, we can show that the JMB on the upper crack surface can be written in Cartesian component form as

$$\sigma_{12} = -\sqrt{(1+u_{1,1})^2 + u_{2,1}^2} \left(\frac{\tilde{\gamma}'(X_1)(1+u_{1,1})^2}{(1+u_{1,1})^2 + u_{2,1}^2} + \frac{\tilde{\gamma}(X_1) u_{2,1} \left[u_{2,1}^2 u_{1,12} + u_{2,1}(1+u_{1,1})(u_{1,11} - u_{2,12}) - (1+u_{1,1})^2 u_{2,11} \right]}{\left((1+u_{1,1})^2 + u_{2,1}^2 \right)^2} \right), \quad (2.6)$$

$$\sigma_{22} = -\sqrt{(1+u_{1,1})^2 + u_{2,1}^2} \left(\frac{\tilde{\gamma}'(X_1)(1+u_{1,1}) u_{2,1}}{(1+u_{1,1})^2 + u_{2,1}^2} - \frac{\tilde{\gamma}(X_1)(1+u_{1,1}) \left[u_{2,1}^2 u_{1,12} + u_{2,1}(1+u_{1,1})(u_{1,11} - u_{2,12}) - (1+u_{1,1})^2 u_{2,11} \right]}{\left((1+u_{1,1})^2 + u_{2,1}^2 \right)^2} \right), \quad (2.7)$$

where $\mathbf{X} \in \Sigma^+$. A similar expression holds for Σ^-. The derivation of these component forms of the JMB is given in Appendix A (cf. Sendova and Walton 2010, equation (10)). These equations will be applied to the FEM model in the form of boundary conditions over the crack surface. However, since these equations are highly nonlinear, we simplify the problem by linearizing these. Sendova and Walton consider two options for the surface tension: either a constant or an expression that depends linearly on curvature. We will discuss the curvature-dependent case in Section 4. For now, we assume a constant surface tension, i.e.,

$$\tilde{\gamma} = \gamma_0, \quad (2.8)$$

where γ_0 is a nondimensional constant.

We apply this constant surface tension to the component-form JMB equations (2.6) and (2.7) and linearize by assuming that $u_{j,k}$ and $u_{i,jk}$ are small whenever $j \neq k$. This yields the linearized JMB equations on the upper crack surface

$$\begin{array}{l} \sigma_{12}(X_1, 0^+) = 0 \\ \sigma_{22}(X_1, 0^+) = -\gamma_0 u_{2,11}(X_1, 0) \end{array}, \ |X_1| \leq 1. \tag{2.9}$$

A similar computation shows that the linearized JMB equations on the lower crack surface are given by

$$\begin{array}{l} \sigma_{12}(X_1, 0^-) = 0 \\ \sigma_{22}(X_1, 0^-) = \gamma_0 u_{2,11}(X_1, 0) \end{array}, \ |X_1| \leq 1. \tag{2.10}$$

In addition to the DMB and JMB governing equations, we make a few assumptions to obtain conditions on the rest of the boundary ($\partial\Omega$) and to write our DMB in terms of the displacement $\mathbf{u}_\kappa$. First, we assume that the body is isotropic and can be modeled constitutively by Hooke's law, i.e.,

$$\mathbf{T}_\kappa = 2\mu\mathbf{E}(\mathbf{u}_\kappa) + \lambda \mathrm{tr}(\mathbf{E}(\mathbf{u}_\kappa))\mathbf{I}, \tag{2.11}$$

where $\mathbf{E}(\mathbf{u}) = \frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^\mathsf{T})$ is the linearized strain tensor and λ and μ are the Lamé moduli (material constants).

We also assume that the sides are traction-free, i.e.,

$$\lim_{X_1 \to \pm\infty} \mathbf{T}_\kappa(X_1, X_2)\mathbf{n} = 0. \tag{2.12}$$

Finally, the far-field tensile loading condition is described by

$$\lim_{X_2 \to \pm\infty} \left\{ \begin{array}{c} \sigma_{11}(X_1, X_2) \\ \sigma_{12}(X_1, X_2) \\ \sigma_{22}(X_1, X_2) \end{array} \right\} = \left\{ \begin{array}{c} 0 \\ 0 \\ \pm\sigma \end{array} \right\}. \tag{2.13}$$

Thus the Griffith crack problem modeled by the Sendova-Walton fracture theory that we consider is described by the system of equations (2.2) and (2.9) – (2.13). Recall, however, that we have nondimensionalized by crack half-length, a, and by Young's modulus, E, which has the same units as stress. Therefore, these equations actually involve the nondimensional quantities

$$\begin{array}{lll} a^\star = \dfrac{a}{a} = 1 & \mathbf{X}^\star = \dfrac{\mathbf{X}}{a} & \mathbf{u}^\star = \dfrac{\mathbf{u}}{a} \\ \mathbf{T}^\star_\kappa = \dfrac{\mathbf{T}_\kappa}{E} & \mathbf{b}^\star_\kappa = \dfrac{\mathbf{b}_\kappa}{E} & \tilde{\gamma}^\star = \dfrac{\tilde{\gamma}}{aE} \\ \sigma^\star = \dfrac{\sigma}{E} & \lambda^\star = \dfrac{\lambda}{E} & \mu^\star = \dfrac{\mu}{E} \end{array} \tag{2.14}$$

where we have simply dropped the $(^\star)$ superscript.

2.2 Well-Posedness

Before we discuss the numerical implementation of the model described above, we must first show that the corresponding variational problem is well-posed. To do so, we take advantage of the fact that the DMB and JMB are now both linear to use superposition to reformulate the problem into an equivalent pure traction problem. This basically results in moving the loading σ from the infinite boundary to the crack surface. In other words, the equivalent pure traction boundary value problem that we consider consists of the same DMB and constitutive equations as before ((2.2)

and (2.11)), and a modified JMB

$$\begin{aligned} &\sigma_{12}(X_1, 0^{\pm}) = 0 \\ &\sigma_{22}(X_1, 0^{\pm}) = \mp\gamma_0 u_{2,11}(X_1, 0) \mp \sigma \end{aligned} \quad , \ |X_1| \leq 1, \tag{2.15}$$

where the entire infinite boundary is now traction-free, i.e.,

$$\lim_{\mathbf{X}\to\pm\infty} \mathbf{T}_\kappa(\mathbf{X})\mathbf{n} = 0. \tag{2.16}$$

The general weak formulation for this problem is found in the standard way, by multiplying by a test function $\mathbf{v}$ and integrating over Ω (see Appendix B for the derivation). This yields

$$a(\mathbf{u}_\kappa, \mathbf{v}) - \int_{\partial\Omega} \mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n} = (\mathbf{v}, \mathbf{b}_\kappa)_\Omega, \tag{2.17}$$

where $a(\cdot, \cdot)$ is the bilinear form

$$a(\mathbf{u}, \mathbf{v}) = \int_\Omega \lambda \operatorname{Div} \mathbf{u} \operatorname{Div} \mathbf{v} + \int_\Omega 2\mu \mathbf{E}(\mathbf{u}) : \mathbf{E}(\mathbf{v}). \tag{2.18}$$

We apply the boundary conditions in (2.15) and (2.16) to determine the value of the boundary term in this weak formulation. The only nonzero contribution occurs over

the crack surface, i.e.,

$$\begin{aligned}\int_{\partial\Omega} \mathbf{v}\cdot\mathbf{T}_\kappa\mathbf{n} &= \int_{\Sigma^\pm} \mathbf{v}\cdot\mathbf{T}_\kappa\mathbf{n}^\mp \\ &= \int_{\Sigma^\pm} \mathbf{v}\cdot\begin{pmatrix}\sigma_{11} & 0\\ 0 & \mp(\gamma_0 u_{2,11}+\sigma)\end{pmatrix}\begin{pmatrix}0\\ \mp 1\end{pmatrix} \\ &= 2\int_{-1}^{1} v_2(\gamma_0 u_{2,11}+\sigma). \end{aligned} \tag{2.19}$$

We further reduce this expression by applying integration by parts to the first term. Since this problem is symmetric about the X_1-axis, and we assume no crack growth, we may assume that the crack tips do not move in the X_2 direction, i.e., $u_2(\pm 1, 0) = 0$. Applying the same assumption to our test function $\mathbf{v}$, we have

$$\begin{aligned}\int_{\partial\Omega} \mathbf{v}\cdot\mathbf{T}_\kappa\mathbf{n} &= 2\gamma_0 v_2(X_1,0)u_{2,1}(X_1,0)\Big|_{-1}^{1} - 2\int_{\Sigma^+}\gamma_0 v_{2,1}u_{2,1} + 2\int_{\Sigma^+}\sigma v_2 \\ &= -\int_\Sigma \gamma_0 u_{2,1}v_{2,1} + \int_\Sigma \sigma v_2. \end{aligned} \tag{2.20}$$

Per convention for pure traction problems (as in Ern and Guermond 2004), we further assume that

$$\int_\Omega \mathbf{u} = \int_\Omega \nabla\times\mathbf{u} = 0. \tag{2.21}$$

Combining (2.17), (2.18), (2.20), and (2.21) yields the following variational problem for the pure traction problem:

Definition 1 (Pure Traction Variational Problem)**.** *Find* $\mathbf{u}\in V$ *such that*

$$A(\mathbf{u},\mathbf{v}) = L(\mathbf{v}), \qquad \forall \quad \mathbf{v}\in V, \tag{2.22}$$

where $A(\cdot,\cdot)$ and $L(\cdot)$ are the bilinear and linear forms, respectively, given by

$$A(\mathbf{u},\mathbf{v}) = a(\mathbf{u},\mathbf{v}) + \int_\Sigma \gamma_0 u_{2,1} v_{2,1}, \tag{2.23}$$

$$L(\mathbf{v}) = (\mathbf{v},\mathbf{b}_\kappa)_\Omega + \int_\Sigma \sigma v_2, \tag{2.24}$$

where $a(\cdot,\cdot)$ is defined in (2.18) and V is the solution and test space given by

$$V := \{\mathbf{u}(\mathbf{X}) \in H^1(\Omega) : u_2(X_1,0) \in H^1(\Sigma),\ \int_\Omega \mathbf{u} = \int_\Omega \nabla\times\mathbf{u} = 0,\ u_2(\pm 1,0) = 0\}. \tag{2.25}$$

We note that V is a Hilbert space with inner product

$$\langle \mathbf{u},\mathbf{v}\rangle_V = \langle \mathbf{u},\mathbf{v}\rangle_{H^1(\Omega)} + \langle u_2, v_2\rangle_{H^1(\Sigma)}, \tag{2.26}$$

and induced norm

$$||\mathbf{u}||_V = \sqrt{\langle u,u\rangle_V}. \tag{2.27}$$

In other words,

$$||\mathbf{u}||_V^2 = ||\mathbf{u}||_{H^1(\Omega)}^2 + ||u_2||_{H^1(\Sigma)}^2, \tag{2.28}$$

where we recall that

$$||\mathbf{u}||_{H^1(\Omega)}^2 = ||\nabla\mathbf{u}||_{L^2(\Omega)}^2 + ||\mathbf{u}||_{L^2(\Omega)}^2, \tag{2.29}$$

and similarly

$$||u_2||_{H^1(\Sigma)}^2 = ||u_{2,1}||_{L^2(\Sigma)}^2 + ||u_2||_{L^2(\Sigma)}^2. \tag{2.30}$$

Next, we will show that the pure traction variational problem is well-posed, i.e., a unique solution exists that depends continuously on the data. This is guaranteed

by the Lax-Milgram lemma (see Grossmann, Roos, and Stynes 2007, p. 145) for our Hilbert space V if we can show that the following three conditions are satisfied:

Condition 1 The bilinear form $A(\cdot,\cdot)$ is continuous in V, i.e.,

$$\exists \;\; M > 0 \;\; \text{s.t.} \;\; |A(\mathbf{u},\mathbf{v})| \leq M||\mathbf{u}||_V||\mathbf{v}||_V \;\; \forall \;\; \mathbf{u},\mathbf{v} \in V.$$

Condition 2 The bilinear form $A(\cdot,\cdot)$ is V-elliptic (or coercive), i.e.,

$$\exists \;\; \alpha > 0 \;\; \text{s.t.} \;\; A(\mathbf{u},\mathbf{u}) \geq \alpha||\mathbf{u}||_V^2 \;\; \forall \;\; \mathbf{u} \in V.$$

Condition 3 The linear form $L(\cdot)$ is continuous in V, i.e.,

$$\exists \;\; m > 0 \;\; \text{s.t.} \;\; |L(\mathbf{v})| \leq m||\mathbf{v}||_V \;\; \forall \;\; \mathbf{v} \in V.$$

Before we show that these conditions hold, we will need the following result.

Proposition 2.1 *For any* $\mathbf{u} \in [L^2(\Omega)]^2$*, we have*

$$||\operatorname{Div}\mathbf{u}||_{L^2(\Omega)} \leq \sqrt{2}||\nabla\mathbf{u}||_{L^2(\Omega)}. \tag{2.31}$$

Proof.

$$\begin{aligned} ||\operatorname{Div}\mathbf{u}||_{L^2(\Omega)}^2 &= \int_\Omega (\operatorname{tr}(\nabla\mathbf{u}))^2 \\ &= \int_\Omega (u_{1,1}+u_{2,2})^2 \\ &= \int_\Omega (u_{1,1}^2+u_{2,2}^2+2u_{1,1}u_{2,2}). \end{aligned} \tag{2.32}$$

Applying the Cauchy-Schwarz inequality to the last term yields

$$|| \operatorname{Div} \mathbf{u}||^2_{L^2(\Omega)} \leq \int_\Omega \left(u^2_{1,1} + u^2_{2,2}\right) + 2 \left(\int_\Omega u^2_{1,1}\right)^{\frac{1}{2}} \left(\int_\Omega u^2_{2,2}\right)^{\frac{1}{2}}. \tag{2.33}$$

We appeal to Young's inequality on the last term to obtain

$$\begin{aligned} || \operatorname{Div} \mathbf{u}||^2_{L^2(\Omega)} &\leq \int_\Omega \left(u^2_{1,1} + u^2_{2,2}\right) + \int_\Omega u^2_{1,1} + \int_\Omega u^2_{2,2} \\ &= \int_\Omega 2\left(u^2_{1,1} + u^2_{2,2}\right) \\ &\leq \int_\Omega 2\left(u^2_{1,1} + u^2_{2,1} + u^2_{1,2} + u^2_{2,2}\right) \\ &= \int_\Omega 2(\nabla \mathbf{u} : \nabla \mathbf{u}) \\ &= 2||\nabla \mathbf{u}||^2_{L^2(\Omega)}. \end{aligned} \tag{2.34}$$

□

Proof of Condition 1. Starting with the bilinear form $A(\cdot, \cdot)$ in (2.23) and applying Cauchy-Schwarz to each term yields

$$\begin{aligned} |A(\mathbf{u}, \mathbf{v})| \leq |\lambda|\, || \operatorname{Div} \mathbf{u}||_{L^2(\Omega)} || \operatorname{Div} \mathbf{v}||_{L^2(\Omega)} + |\mu|\, ||\nabla \mathbf{u}||_{L^2(\Omega)} ||\nabla \mathbf{v}||_{L^2(\Omega)} \\ + |\mu|\, ||\nabla \mathbf{u}^\mathsf{T}||_{L^2(\Omega)} ||\nabla \mathbf{v}||_{L^2(\Omega)} + |\gamma_0|\, ||u_{2,1}||_{L^2(\Sigma)} ||v_{2,1}||_{L^2(\Sigma)}. \end{aligned} \tag{2.35}$$

We note that $\nabla \mathbf{u}^\mathsf{T} : \nabla \mathbf{u}^\mathsf{T} = \nabla \mathbf{u} : \nabla \mathbf{u}$ to combine the second and third terms on the right-hand side. We also apply Proposition 2.1 to obtain

$$\begin{aligned} |A(\mathbf{u}, \mathbf{v})| &\leq 2|\lambda|\, ||\nabla \mathbf{u}||_{L^2(\Omega)} ||\nabla \mathbf{v}||_{L^2(\Omega)} + 2|\mu|\, ||\nabla \mathbf{u}||_{L^2(\Omega)} ||\nabla \mathbf{v}||_{L^2(\Omega)} \\ &\quad + |\gamma_0|\, ||u_{2,1}||_{L^2(\Sigma)} ||v_{2,1}||_{L^2(\Sigma)} \\ &= 2(|\lambda| + |\mu|) ||\nabla \mathbf{u}||_{L^2(\Omega)} ||\nabla \mathbf{v}||_{L^2(\Omega)} + |\gamma_0|\, ||u_{2,1}||_{L^2(\Sigma)} ||v_{2,1}||_{L^2(\Sigma)}. \end{aligned} \tag{2.36}$$

Next, we notice from (2.28) – (2.30) that any single term of the V-norm cannot exceed the whole, since all the terms are non-negative, e.g.,

$$||\nabla \mathbf{u}||_{L^2(\Omega)} \leq ||\mathbf{u}||_{H^1(\Omega)} \leq ||\mathbf{u}||_V. \tag{2.37}$$

Applying this result to the previous equation yields

$$|A(\mathbf{u}, \mathbf{v})| \leq M||\mathbf{u}||_V||\mathbf{v}||_V, \tag{2.38}$$

where

$$M = 2(|\lambda| + |\mu|) + |\gamma_0|. \tag{2.39}$$

□

Proof of Condition 2. To show coercivity, we first note that the shear modulus μ is always positive. We assume the typical range of Poisson's ratio for a compressible material, i.e., $0 \leq \nu < \frac{1}{2}$, which implies that $\lambda \geq 0$. We also assume a non-negative surface tension, i.e., $\gamma_0 \geq 0$. Ern and Guermond (2004) show, using Korn's Second Inequality and the Petree-Tartar Lemma, that there exists a constant $c_0 > 0$ such that

$$c_0||\mathbf{u}||_{H^1(\Omega)} \leq ||\mathbf{E}(\mathbf{u})||_{L^2(\Omega)}, \quad \forall \ \mathbf{u} \in V. \tag{2.40}$$

This shows that $a(\cdot, \cdot)$ is H^1-elliptic over Ω, since

$$\begin{aligned} a(\mathbf{u}, \mathbf{u}) &= \lambda||\operatorname{Div} \mathbf{u}||^2_{L^2(\Omega)} + 2\mu||\mathbf{E}(\mathbf{u})||^2_{L^2(\Omega)} \\ &\geq 0 + 2\mu\, c_0^2\, ||\mathbf{u}||^2_{H^1(\Omega)}. \end{aligned} \tag{2.41}$$

In addition, by the Poincaré-Friedrichs Inequality (see Ern and Guermond 2004, p. 491), there exists a constant $c_1 > 0$ such that

$$c_1||u_2||_{H^1(\Sigma)} \leq ||u_{2,1}||_{L^2(\Sigma)}, \quad \forall \ \mathbf{u} \in V. \tag{2.42}$$

Combining these two results, we see that $A(\cdot,\cdot)$ is V-elliptic, since

$$\begin{aligned} A(\mathbf{u},\mathbf{u}) &= a(\mathbf{u},\mathbf{u}) + \gamma_0||u_{2,1}||^2_{L^2(\Sigma)} \\ &\geq 2\mu\, c_0^2\, ||\mathbf{u}||^2_{H^1(\Omega)} + \gamma_0\, c_1^2\, ||u_2||^2_{H^1(\Sigma)} \\ &\geq \alpha||\mathbf{u}||^2_V, \end{aligned} \tag{2.43}$$

where

$$\alpha = \min\{2\mu\, c_0^2, \gamma_0\, c_1^2\}. \tag{2.44}$$

□

Proof of Condition 3. To show that the linear form $L(\cdot)$ in (2.24) is continuous in V, we again apply Cauchy-Schwarz to both terms. This yields

$$|L(\mathbf{v})| \leq ||\mathbf{v}||_{L^2(\Omega)}||\mathbf{b}_\kappa||_{L^2(\Omega)} + |\sigma| \left(\int_\Sigma 1^2\right)^{\frac{1}{2}} \left(\int_\Sigma v_2^2\right)^{\frac{1}{2}}. \tag{2.45}$$

We note that

$$\begin{aligned} \int_\Sigma 1 &= \int_{\Sigma^+} 1 + \int_{\Sigma^-} 1 \\ &= x\Big|_{-1}^{1} + x\Big|_{-1}^{1} \\ &= 4, \end{aligned} \tag{2.46}$$

which implies

$$\begin{aligned} |L(\mathbf{v})| &\leq ||\mathbf{b}_\kappa||_{L^2(\Omega)}||\mathbf{v}||_{L^2(\Omega)} + 2|\sigma|\,||v_2||_{L^2(\Sigma)} \\ &\leq ||\mathbf{b}_\kappa||_{L^2(\Omega)}||\mathbf{v}||_V + 2|\sigma|\,||\mathbf{v}||_V \\ &= m\,||\mathbf{v}||_V, \end{aligned} \tag{2.47}$$

where

$$m = 2|\sigma| + ||\mathbf{b}_\kappa||_{L^2(\Omega)}. \tag{2.48}$$

□

Thus, by the Lax-Milgram Lemma, we have shown that the pure traction variational problem is well-posed. Since this problem is equivalent to the original boundary value problem we posed in Section 2.1, this problem is also well-posed. We will discuss the numerical implementation of this problem in the next section.

3. NUMERICAL IMPLEMENTATION WITH CONSTANT SURFACE TENSION

For the numerical implementation of the Sendova-Walton model applied to the Griffith crack problem in the case of constant surface tension, as described in Section 2.1, we make a few simplifications. First, we assume that the mutual body force $\mathbf{b}_\kappa$ is zero. We choose a zero body force since Sendova and Walton show that it is not required to obtain a finite opening angle and a reduction to a logarithmic singularity in the crack-tip stress. However, if a nonzero body force is desired, we note that it is straightforward to add into our implementation.

Second, we take advantage of the fact that the displacement solution of this strictly mode-I problem is symmetric about both the X_1 and X_2 axes. This allows us to reduce the domain from the entire plane to the upper right quarter-plane. However, since we cannot actually implement the FEM on an infinite body, we approximate the upper right quarter-plane by the finite square domain $Q = [0, b]^2$, where b is the body half-length. Recall that we have nondimensionalized the system both by Young's modulus E and by crack half-length a. We may then take the (dimensionless) body half-length b to be as large as desired relative to the unit crack half-length to approximate the infinite domain problem.

The finite computational domain Q is shown in Figure 3.1 below, where $\Gamma_{(\cdot)}$ indicates the piece of the boundary ∂Q of Q corresponding to the **T**op, **L**eft, or **R**ight faces, the **C**rack surface, or the rest of the **B**ottom face outside the crack, respectively. Note that Γ_C indicates the right-hand side of the upper crack surface, which lies along the bottom face of Q so that the center point of the upper crack surface is at the origin.

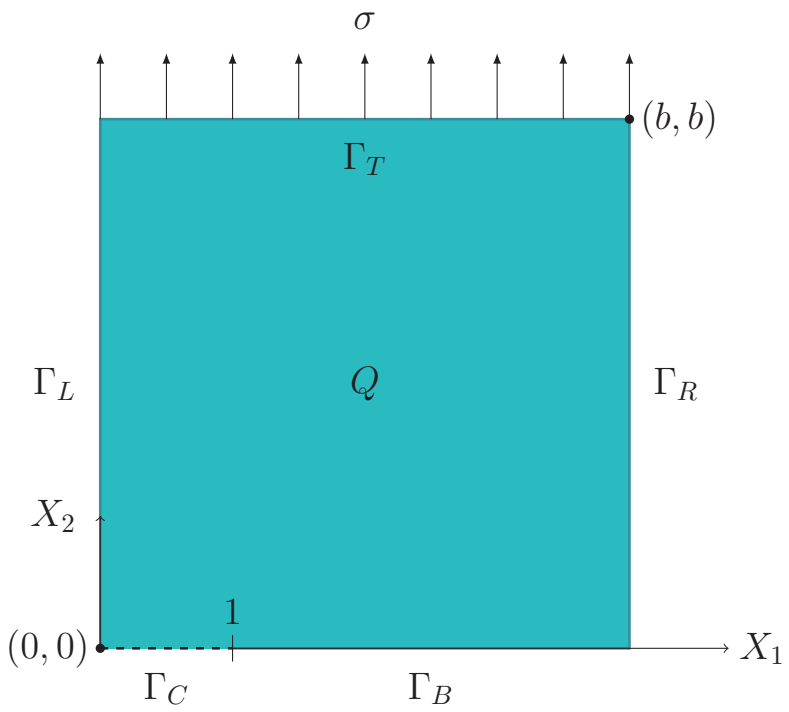


Figure 3.1 Finite square domain $Q = [0, b]^2$ which approximates the quarter-plane

This added symmetry results in some new boundary conditions. The top face is still subjected to the uniform tensile loading σ, as in (2.13), but the bottom face Γ_B now satisfies the symmetry boundary condition

$$\left.\begin{array}{r} u_2(X_1, 0) = 0 \\ \sigma_{12}(X_1, 0) = 0 \end{array}\right\}, \quad \text{for } 1 \leq X_1 \leq b. \tag{3.1}$$

Similarly, the right-hand side Γ_R is still traction-free, but new symmetry boundary conditions are applied to the left-hand side, i.e.,

$$\left.\begin{array}{r} u_1(0, X_2) = 0 \\ u_{2,1}(0, X_2) = 0 \end{array}\right\}, \quad \text{for } 0 \leq X_2 \leq b. \tag{3.2}$$

The corresponding general weak formulation for this problem is identical to that of the infinite plane problem, i.e.,

$$a(\mathbf{u}_\kappa, \mathbf{v}) - \int_{\partial Q} \mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n} = (\mathbf{v}, \mathbf{b}_\kappa)_Q, \tag{3.3}$$

where $a(\cdot, \cdot)$ is the bilinear form

$$a(\mathbf{u}, \mathbf{v}) = \int_Q \lambda \operatorname{Div} \mathbf{u} \operatorname{Div} \mathbf{v} + \int_Q 2\mu \mathbf{E}(\mathbf{u}) : \mathbf{E}(\mathbf{v}). \tag{3.4}$$

We again apply the boundary conditions to determine the value of the boundary term in this formulation. In particular, we have

$$\int_{\partial Q} \mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n} = \left[\int_{\Gamma_T} + \int_{\Gamma_B} + \int_{\Gamma_R} + \int_{\Gamma_L} + \int_{\Gamma_C} \right] \mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n}. \tag{3.5}$$

For the integral over the top face, recall from (2.13) that the loading condition is given by

$$\mathbf{T}_\kappa(\mathbf{X}) = \begin{pmatrix} 0 & 0 \\ \\ 0 & \sigma \end{pmatrix}, \quad \text{for } \mathbf{X} \in \Gamma_T. \tag{3.6}$$

Thus we have

$$\mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n} \Big|_{\Gamma_T} = \mathbf{v} \cdot \begin{pmatrix} 0 & 0 \\ \\ 0 & \sigma \end{pmatrix} \begin{pmatrix} 0 \\ \\ 1 \end{pmatrix} = \sigma v_2. \tag{3.7}$$

We apply the symmetry boundary conditions (3.1) to the bottom edge of Q excluding the crack surface. We assume that the test function $\mathbf{v}$ satisfies the same

homogeneous Dirichlet condition on this piece of the boundary, i.e.,

$$v_2 \Big|_{\Gamma_B} = 0. \tag{3.8}$$

This yields

$$\mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n} \Big|_{\Gamma_B} = \begin{pmatrix} v_1 \\ \\ 0 \end{pmatrix} \cdot \begin{pmatrix} \sigma_{11} & 0 \\ \\ 0 & \sigma_{22} \end{pmatrix} \begin{pmatrix} 0 \\ \\ -1 \end{pmatrix} = 0. \tag{3.9}$$

The right-hand side is still traction free, i.e.,

$$\mathbf{T}_\kappa(\mathbf{X})\mathbf{n} = 0, \quad \forall \ \mathbf{X} \in \Gamma_R, \tag{3.10}$$

from which we immediately see that the corresponding integral over this face is zero.

For the left-hand side, we first note that the second condition in (3.2) is equivalent to

$$\sigma_{12}(0, X_2) = 0, \quad 0 \leq X_2 \leq b, \tag{3.11}$$

since writing Hooke's law (2.11) in component form (obtained by applying the definition of the linearized strain tensor $\mathbf{E}(\mathbf{u}_\kappa)$) yields

$$\begin{aligned} \sigma_{12}(0, X_2) &= \mu \Big[u_{1,2}(0, X_2) + u_{2,1}(0, X_2) \Big] \\ &= \mu u_{2,1}(0, X_2), \end{aligned} \tag{3.12}$$

where $u_{1,2}(0, X_2)$ vanishes by the first condition in (3.2). Applying this equivalent boundary condition to the left face yields

$$\mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n} \Big|_{\Gamma_L} = \begin{pmatrix} 0 \\ \\ v_2 \end{pmatrix} \cdot \begin{pmatrix} \sigma_{11} & 0 \\ \\ 0 & \sigma_{22} \end{pmatrix} \begin{pmatrix} -1 \\ \\ 0 \end{pmatrix} = 0, \tag{3.13}$$

where we have again assumed that $\mathbf{v}$ satisfies the same Dirichlet condition as $\mathbf{u}_\kappa$.

Finally, we apply the same linearized JMB equations (2.9) to the (upper) crack surface to obtain

$$\begin{aligned} \mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n} \Big|_{\Gamma_C} &= \begin{pmatrix} v_1 \\ \\ v_2 \end{pmatrix} \cdot \begin{pmatrix} \sigma_{11} & 0 \\ \\ 0 & -\gamma_0 u_{2,11} \end{pmatrix} \begin{pmatrix} 0 \\ \\ -1 \end{pmatrix} \\ &= \gamma_0 v_2 u_{2,11}. \end{aligned} \tag{3.14}$$

The integral over the crack surface can be further simplified by applying integration by parts, which yields

$$\begin{aligned} \int_{\Gamma_C} \mathbf{v} \cdot \mathbf{T}_\kappa \mathbf{n} &= \int_0^1 \gamma_0 v_2 u_{2,11} \\ &= \gamma_0 v_2(X_1, 0) u_{2,1}(X_1, 0) \Big|_0^1 - \int_{\Gamma_C} \gamma_0 v_{2,1} u_{2,1} \\ &= - \int_{\Gamma_C} \gamma_0 v_{2,1} u_{2,1}, \end{aligned} \tag{3.15}$$

since the endpoint terms vanish by (3.2) and (3.8).

We combine (3.7), (3.9), (3.10), (3.13), and (3.15) with the general weak formulation in (3.3) to obtain the corresponding variational problem on the finite computational domain Q in the case of constant surface tension:

Definition 2 (Variational Problem on Q)**.** *Find* $\mathbf{u} \in V_Q$ *such that*

$$A(\mathbf{u}, \mathbf{v}) = L(\mathbf{v}), \qquad \forall \quad \mathbf{v} \in V_Q, \tag{3.16}$$

where $A(\cdot, \cdot)$ *and* $L(\cdot)$ *are the bilinear and linear forms, respectively, given by*

$$A(\mathbf{u}, \mathbf{v}) = a(\mathbf{u}, \mathbf{v}) + \int_{\Gamma_C} \gamma_0 u_{2,1} v_{2,1}, \tag{3.17}$$

$$L(\mathbf{v}) = (\mathbf{v}, \mathbf{b}_\kappa)_Q + \int_{\Gamma_T} \sigma v_2, \tag{3.18}$$

where $a(\cdot, \cdot)$ *is defined in (3.4) and* V_Q *is the solution and test space given by*

$$V_Q := \left\{ \mathbf{u}(\mathbf{X}) \in H^1(Q) : u_2(X_1, 0) \in H^1(\Gamma_C),\, u_2 \Big|_{\Gamma_B} = u_1 \Big|_{\Gamma_L} = 0 \right\}. \tag{3.19}$$

3.1 Finite Element Implementation

We used the finite element method (FEM) to solve the constant surface tension variational problem in Definition 2 using standard bilinear elements. The program we developed employs the `deal.II` finite element library (Bangerth, Hartmann, and Kanschat 2007, 2012) and is similar to the step-8 and step-18 `deal.II` tutorial programs, both of which solve a problem in elasticity. For the main program, which executes the quasi-static brittle fracture simulation, we create the class `ConstST_Fracture` whose member routines execute the major steps of the program, which are summarized by

1. Load the run-time parameters from the parameter file `ConstST_Fracture.prm`. These data include the far-field tensile loading, the constant surface tension parameter γ_0, and the Lamé constants.

2. Create a coarse triangulation of the rectangular domain. The initial mesh is square uniform with top, left, right, bottom, and crack surfaces indicated on the boundary.

3. Set up the bilinear finite element and the corresponding degrees of freedom.

4. Assemble the system. Use the symmetric stress-strain tensor to assemble the terms in the stiffness matrix. Add in the contribution of the term on the crack surface due to the JMB condition.

5. Solve the system using the conjugate gradient method, since the matrix is symmetric.

6. Output the results. These include the displacement and norm of strain in the entire body, the crack opening profile (u_1, u_2), the profile slope $u_{2,1}$ along the crack surface, and the stress component σ_{22} along the bottom face outside the crack.

7. Adaptively refine (and coarsen) the grid and repeat the process.

We note here that we actually allow two choices for the finite computational domain: either the square quadrant domain as shown in Figure 3.1 or a similar bar domain $B = [0, b] \times [0, 1]$ which is used to reduce the total degrees of freedom. The picture for the bar domain is identical to that of Figure 3.1 if one replaces the top right point (b, b) with the point $(b, 1)$.

Along with the main fracture code, we developed two helper classes that aid in the implementation. The first, `CreateUCD` allows us to have more control on the definition of the initial mesh, which is a coarse uniform mesh. We also developed some new routines in `SaveData` that simplify the process of saving meshes and data sets.

Finally, we use some classes originally developed by M. Sebastian Pauletti that contain grid manipulation tools, including a routine that deforms a grid by the solution vector. The entire code along with all the required helper files may be found in Appendix C.

3.2 Results of the Brittle Fracture Code with Constant Surface Tension

For our model, we assume an idealized brittle material, much like silicon. We cannot say that we model actual silicon, since it is an anisotropic material and we assume isotropy in our theory. However, we will assume that our material is silicon-like, in particular, we will use the same mechanical properties of silicon. It requires only two independent elastic constants to characterize an isotropic, linear elastic material. Commonly used pairs are Young's modulus (E) and Poisson's ratio (ν); shear modulus (μ) and Poisson's ratio; and shear modulus and bulk modulus (k). We choose the first of these options and find the shear modulus μ and the Lamé elastic constant λ, that appear in Hooke's law (2.11) in terms of E and ν. However, we nondimensionalized the system by Young's modulus, so in practice we require only Poisson's ratio to describe our material.

Due to the anisotropy of silicon, the measured values of these elastic constants have a wide range depending on the orientation of the measuring device. We found that the typical value of ν for silicon ranges anywhere from 0.22 to 0.28. To fix a specific value, we considered the mechanical properties data given in (Sikora 2012; Ioffe Physico-Technical Institute 2001; Vandenberghe 1999). These agreed most closely on the values of the bulk and shear moduli as $k = 98$ GPa and $\mu = 64.1$ GPa, respectively. Using these values, we computed the remaining elastic constants from the relational formulas in (Sadd 2009, p. 85, Table 4-1). Table 3.1 summarizes the resulting mechanical constants that we use for our model.

Table 3.1 Mechanical constants for Si

Mechanical Constant	Relation	Value
Bulk Modulus (k)	k	98 GPa
Shear Modulus (μ)	μ	64.1 GPa
Poisson's Ratio (ν)	$\nu = \dfrac{3k - 2\mu}{6k + 2\mu}$	0.2315
Nondimensionalized λ	$\lambda^\star = \dfrac{\nu}{(1+\nu)(1-2\nu)}$	.3501
Nondimensionalized μ	$\mu^\star = \dfrac{1}{2(1+\nu)}$	.406

We executed the program described in Section 3.1 using the parameters given in Table 3.1. We take the far-field tensile loading σ to be between 0.005 and 0.05. Here, we have taken the upper limit of σ to be the approximate yield strength of silicon. Davis (2004) notes that the yield strength in metals can be defined as the stress required to produce a particular total strain. We use this definition and specify a total strain of 0.05, or 5% elongation. Since we nondimensionalize by Young's modulus, the corresponding yield stress is exactly 0.05. We vary the constant surface tension γ_0 from zero, which would correspond to a LEFM crack, to ten. We typically solved on the bar domain with three cells, i.e., with body half-length $b = 3$, in order to limit the total number of degrees of freedom (DOF). In particular, Table 3.2 shows the average number of total cells, total DOF, and number of cells over the crack surface used for a bar domain with $b = 3$. The zeroth refinement cycle refers to the initial coarse mesh of the bar domain, which contains exactly three cells. We typically ran the program for 20 refinement cycles, for which the final cycle has more than a thousand cells along the crack surface.

Table 3.2 Average number of cells and DOF for each refinement cycle for the bar domain with $b = 3$

Cycle	Total Cells	Total DOF	Crack Cells
0	3	16	1
1	9	34	2
2	21	68	4
3	48	137	6
4	102	266	10
5	208	516	16
6	394	955	19
7	765	1,788	31
8	1,432	3,307	42
9	2,718	6,098	58
10	5,026	11,207	79
11	9,481	20,656	110
12	17,358	37,796	151
13	32,303	69,229	210
14	58,913	126,359	288
15	108,000	228,133	379
16	196,865	415,610	534
17	355,275	742,114	725
18	648,299	1,349,209	953
19	1,162,829	2,401,439	1,293
20	2,278,464	4,330,009	1,730

Although the bar domain certainly allows us to reduce the run time and computational expense, it may not be quite as accurate as the quadrant domain. We compared crack profiles for both domains using body half-lengths of 3, 10, and 100, respectively, in Figures 3.2 and 3.3 below. We used $\sigma = 0.05$ for both zero and nonzero values of γ_0, where the profile was obtained by deforming the original crack surface by the displacement solution vector (in this case, after the 10th refinement cycle). We find

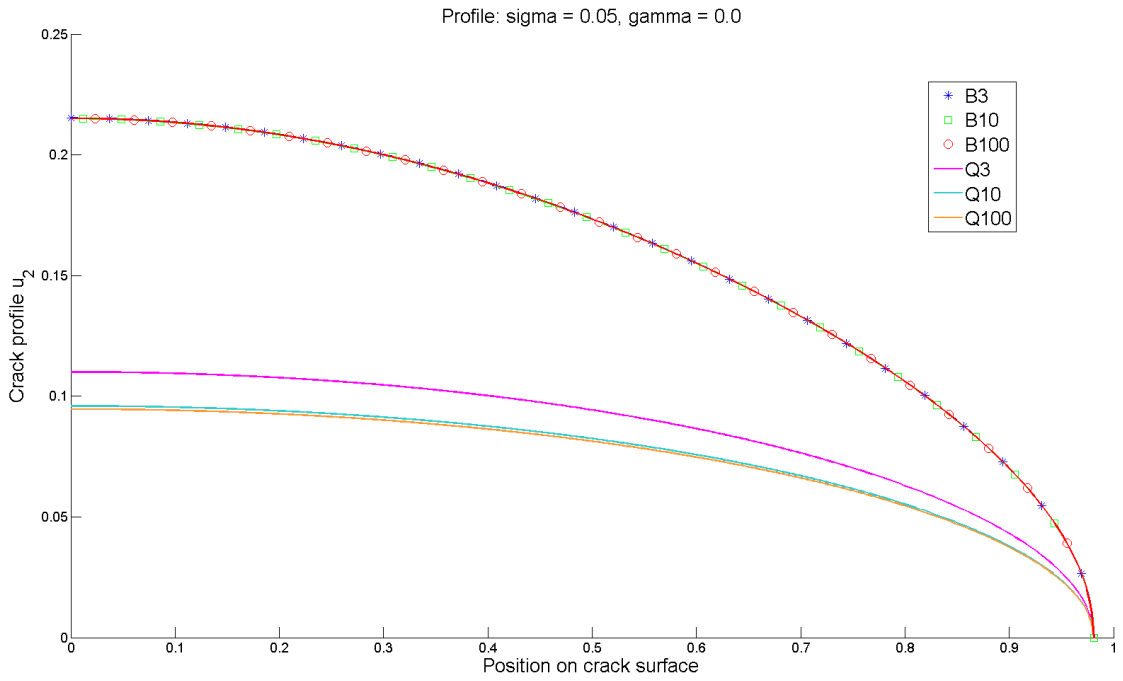


Figure 3.2 Comparison of crack profiles for different domains for $\sigma = 0.05$, $\gamma_0 = 0.0$

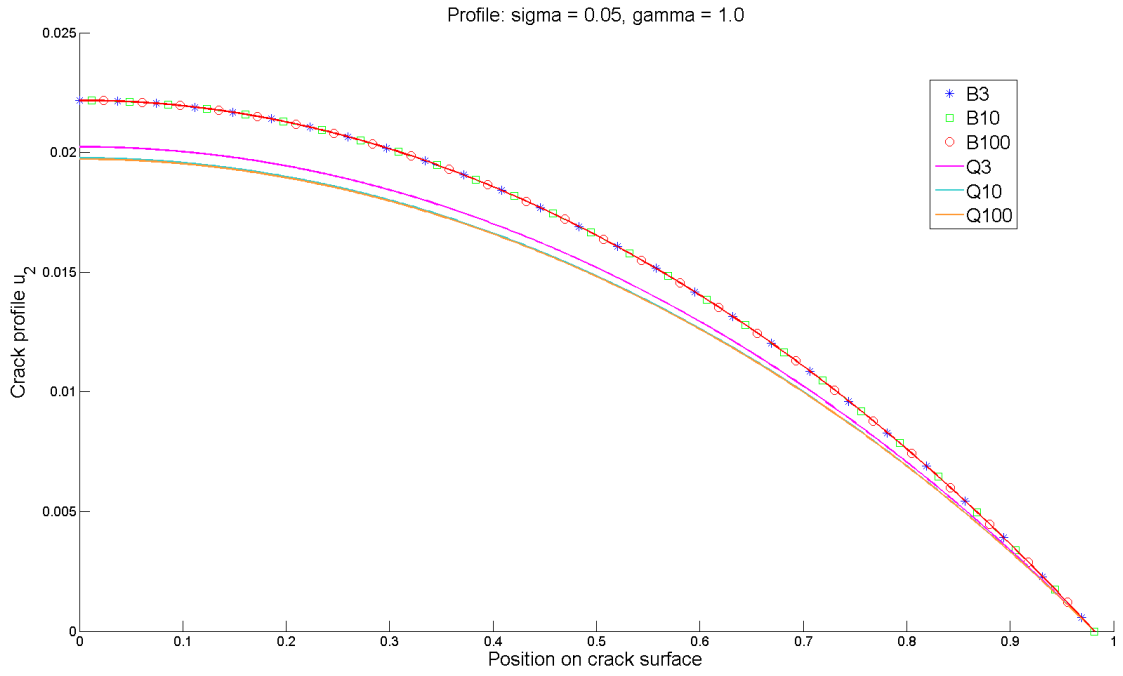


Figure 3.3 Comparison of crack profiles for different domains for $\sigma = 0.05$, $\gamma_0 = 1.0$

that the profiles for the bar domain are nearly identical for the various half-lengths. Similarly, we see that we may only need a half-length of approximately 10 to obtain sufficiently accurate results for the quadrant domain. This is encouraging since we want to approximate the far-field loading situation by taking the body half-length to be sufficiently large, and these results show that the required value of b may be quite small. It makes sense that quadrant domain would do a better job of approximating the far-field loading condition than the bar domain, for which the distance to the far-field loading is same as the crack half-length. Indeed we see that applying the loading close to the crack (the bar domain) produces more deformation than when applied farther away (the quadrant domain).

One of our future goals is to compare these results with experimental data, for which it will certainly be more appropriate to use the quadrant domain, or perhaps a similar vertical bar. However, for the purposes of this dissertation, we restrict ourselves to the bar domain, noting that for the 10th refinement cycle using $b = 100$, solving on the quadrant took more than 8 million DOF, compared with only about 125,000 DOF on the bar. The results with the bar domain are still reasonably close to those of the quadrant, especially for smaller values of σ or larger values of γ_0. More importantly, we will see below they also show the same trends in the data predicted by the Sendova-Walton model.

Before we look at these trends, we first want to show how the solution converges across the refinement cycles. Tables 3.3 – 3.5 show how the center node converges as the mesh is refined for $\sigma = 0.005, 0.025$, and 0.05, respectively. The center node displacement is only given in the X_2-direction, since by symmetry about the X_2-axis there is no displacement in the X_1-direction. We see consistent convergence of the center node displacement in these tables, with accuracy to five or six significant digits after approximately 15 cycles.

Table 3.3 Convergence of the center node displacement across refinement cycles for $\sigma = 0.005$

Cycle	$\gamma_0 = 0.0$	$\gamma_0 = 0.001$	$\gamma_0 = 0.01$	$\gamma_0 = 0.1$	$\gamma_0 = 1.0$	$\gamma_0 = 10.0$
0	0.0117012	0.0116467	0.011178	0.00797059	0.0020599	0.00024477
1	0.0151498	0.0150589	0.0142871	0.00944076	0.00214665	0.000245953
2	0.0171552	0.0170328	0.0160101	0.0100847	0.00217501	0.000246313
3	0.0193603	0.0191962	0.017853	0.0107194	0.00220114	0.000246642
4	0.0202412	0.0200497	0.0185206	0.0109151	0.00220917	0.000246741
5	0.0208774	0.0206598	0.0189717	0.0110054	0.00221298	0.000246788
6	0.0211568	0.0209168	0.0191584	0.011074	0.00221521	0.000246812
7	0.0213386	0.0210896	0.0192646	0.0111069	0.00221617	0.000246826
8	0.0214378	0.021169	0.0193144	0.0111219	0.00221668	0.000246834
9	0.0214921	0.021211	0.0193423	0.0111298	0.002217	0.000246837
10	0.0215157	0.0212284	0.0193539	0.0111342	0.00221718	0.000246839
11	0.0215311	0.0212398	0.0193622	0.0111368	0.00221727	0.00024684
12	0.0215377	0.0212437	0.0193651	0.0111378	0.00221731	0.000246841
13	0.0215423	0.0212469	0.0193676	0.0111386	0.00221734	0.000246841
14	0.0215442	0.0212481	0.0193685	0.0111388	0.00221735	0.000246841
15	0.0215455	0.021249	0.0193692	0.0111391	0.00221736	0.000246842
16	0.0215459	0.0212493	0.0193694	0.0111392	0.00221736	0.000246842
17	0.0215463	0.0212495	0.0193696	0.0111392	0.00221737	0.000246842
18	0.0215465	0.0212497	0.0193697	0.0111393	0.00221737	0.000246842
19	0.0215466	0.0212497	0.0193698	0.0111393	0.00221737	0.000246842
20	0.0215467	0.0212498	0.0193698	0.0111393	0.00221737	0.000246842

Table 3.4 Convergence of the center node displacement across refinement cycles for $\sigma = 0.025$

Cycle	$\gamma_0 = 0.0$	$\gamma_0 = 0.001$	$\gamma_0 = 0.01$	$\gamma_0 = 0.1$	$\gamma_0 = 1.0$	$\gamma_0 = 10.0$
0	0.058506	0.0582335	0.0558901	0.0398529	0.0102995	0.00122385
1	0.0757491	0.0752945	0.0714354	0.0472038	0.0107333	0.00122976
2	0.0857758	0.0851642	0.0800506	0.0504236	0.0108751	0.00123156
3	0.0968014	0.0959811	0.089265	0.0535972	0.0110057	0.00123321
4	0.101206	0.100249	0.0926032	0.0545753	0.0110458	0.0012337
5	0.104387	0.103299	0.0948586	0.0550269	0.0110649	0.00123394
6	0.105784	0.104584	0.0957918	0.05537	0.011076	0.00123406
7	0.106693	0.105448	0.0963229	0.0555344	0.0110809	0.00123413
8	0.107189	0.105845	0.0965722	0.0556095	0.0110834	0.00123417
9	0.107461	0.106055	0.0967117	0.0556492	0.011085	0.00123419
10	0.107578	0.106142	0.0967697	0.0556712	0.0110859	0.0012342
11	0.107656	0.106199	0.096811	0.0556838	0.0110863	0.0012342
12	0.107689	0.106218	0.0968257	0.0556889	0.0110866	0.00123421
13	0.107712	0.106235	0.096838	0.0556928	0.0110867	0.00123421
14	0.107721	0.10624	0.0968423	0.0556942	0.0110868	0.00123421
15	0.107727	0.106245	0.0968458	0.0556954	0.0110868	0.00123421
16	0.10773	0.106246	0.0968471	0.0556958	0.0110868	0.00123421
17	0.107732	0.106248	0.0968481	0.0556962	0.0110868	0.00123421
18	0.107733	0.106248	0.0968485	0.0556963	0.0110868	0.00123421
19	0.107733	0.106249	0.0968488	0.0556964	0.0110868	0.00123421
20	0.107733	0.106249	0.0968489	0.0556964	0.0110868	0.00123421

Table 3.5 Convergence of the center node displacement across refinement cycles for $\sigma = 0.05$

Cycle	$\gamma_0 = 0.0$	$\gamma_0 = 0.001$	$\gamma_0 = 0.01$	$\gamma_0 = 0.1$	$\gamma_0 = 1.0$	$\gamma_0 = 10.0$
0	0.117012	0.116467	0.11178	0.0797059	0.020599	0.0024477
1	0.151498	0.150589	0.142871	0.0944076	0.0214665	0.00245953
2	0.171552	0.170328	0.160101	0.100847	0.0217501	0.00246313
3	0.193603	0.191962	0.17853	0.107194	0.0220114	0.00246642
4	0.202412	0.200497	0.185206	0.109151	0.0220917	0.00246741
5	0.208774	0.206598	0.189717	0.110054	0.0221298	0.00246788
6	0.211568	0.209168	0.191584	0.11074	0.0221521	0.00246812
7	0.213386	0.210896	0.192646	0.111069	0.0221617	0.00246826
8	0.214378	0.21169	0.193144	0.111219	0.0221668	0.00246834
9	0.214921	0.21211	0.193423	0.111298	0.02217	0.00246837
10	0.215157	0.212284	0.193539	0.111342	0.0221718	0.00246839
11	0.215311	0.212398	0.193622	0.111368	0.0221727	0.0024684
12	0.215377	0.212437	0.193651	0.111378	0.0221731	0.00246841
13	0.215423	0.212469	0.193676	0.111386	0.0221734	0.00246841
14	0.215442	0.212481	0.193685	0.111388	0.0221735	0.00246841
15	0.215455	0.21249	0.193692	0.111391	0.0221736	0.00246842
16	0.215459	0.212493	0.193694	0.111392	0.0221736	0.00246842
17	0.215463	0.212495	0.193696	0.111392	0.0221737	0.00246842
18	0.215465	0.212497	0.193697	0.111393	0.0221737	0.00246842
19	0.215466	0.212497	0.193698	0.111393	0.0221737	0.00246842
20	0.215467	0.212498	0.193698	0.111393	0.0221737	0.00246842

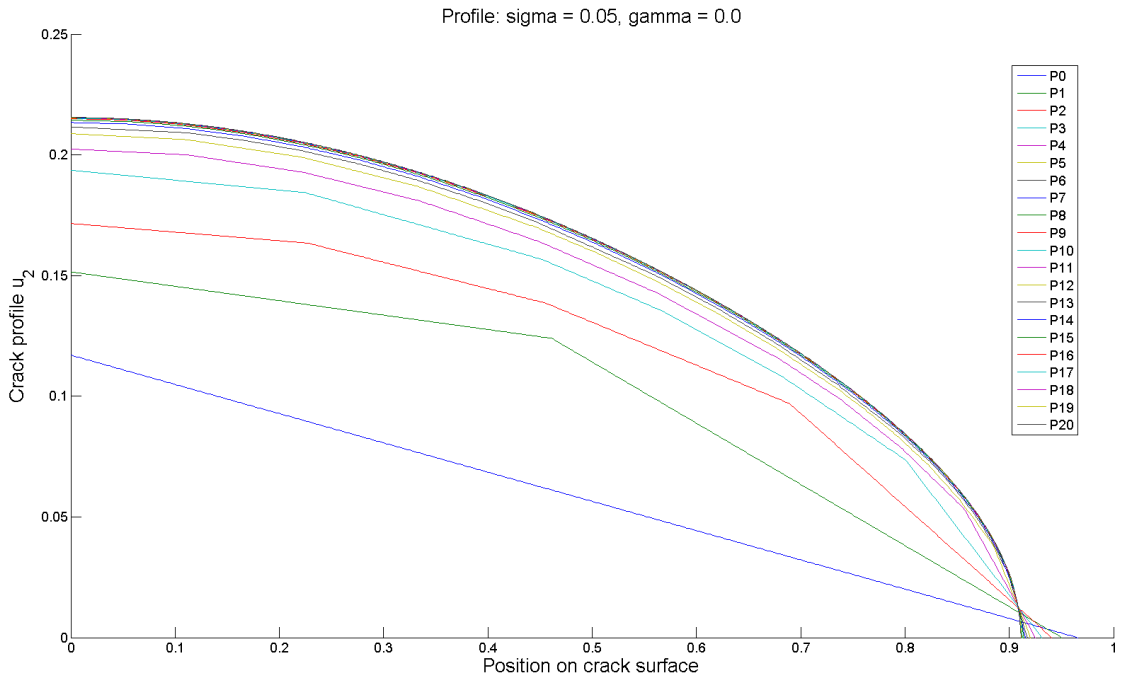


Figure 3.4 Convergence of crack profiles for $\sigma = 0.05$, $\gamma_0 = 0.0$

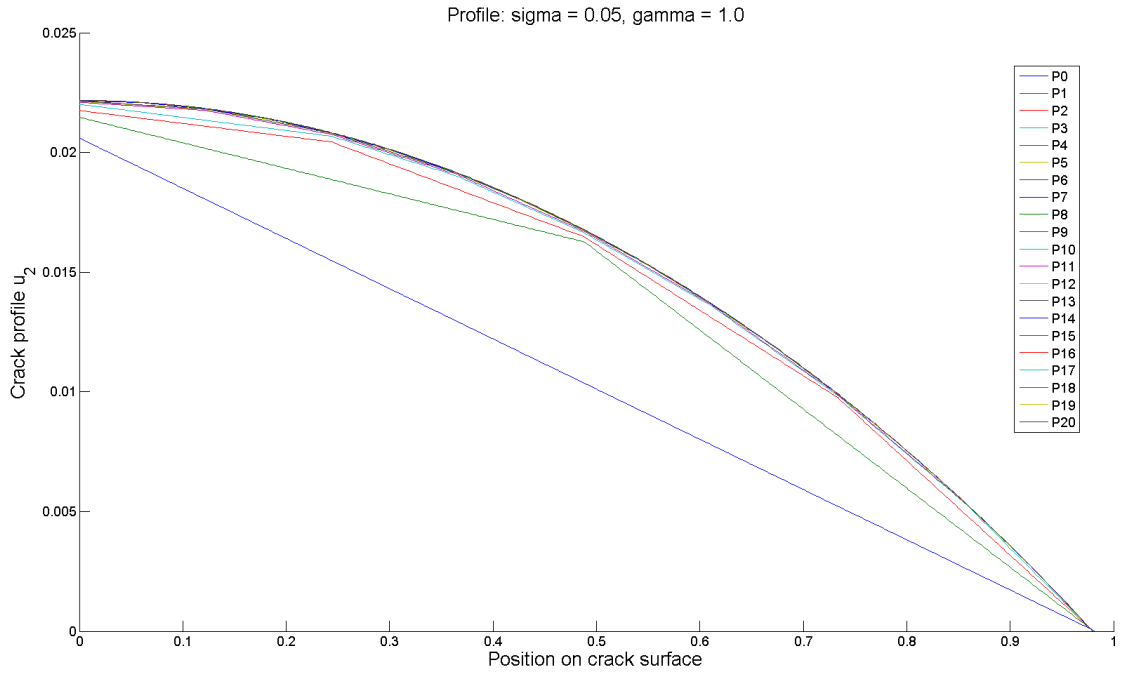


Figure 3.5 Convergence of crack profiles for $\sigma = 0.05$, $\gamma_0 = 1.0$

In addition, Figures 3.4 and 3.5 show the convergence of the entire profile for $\sigma = 0.05$ and $\gamma_0 = 0.0$ and 1.0, respectively, across the refinement cycles. The profiles for other values of σ and γ_0 converge similarly. From Tables 3.3 – 3.5 we also notice that the displacement decreases as the constant surface tension parameter γ_0 increases. This is exactly what we expect, since surface tension acts in opposition to the tensile loading. We see that this holds true for the entire crack profile, not just the center node, in Figures 3.6 – 3.8 which show the crack profile for increasing values of γ_0 for $\sigma = 0.005, 0.025$, and 0.05, respectively. The profiles for these, and the remaining figures in this section, were generated from the displacement solution vector after the 20th refinement cycle.

We also want to compare crack profiles for fixed surface tension as the loading increases. Figure 3.9 shows six sets of crack profiles, each for a different fixed value of

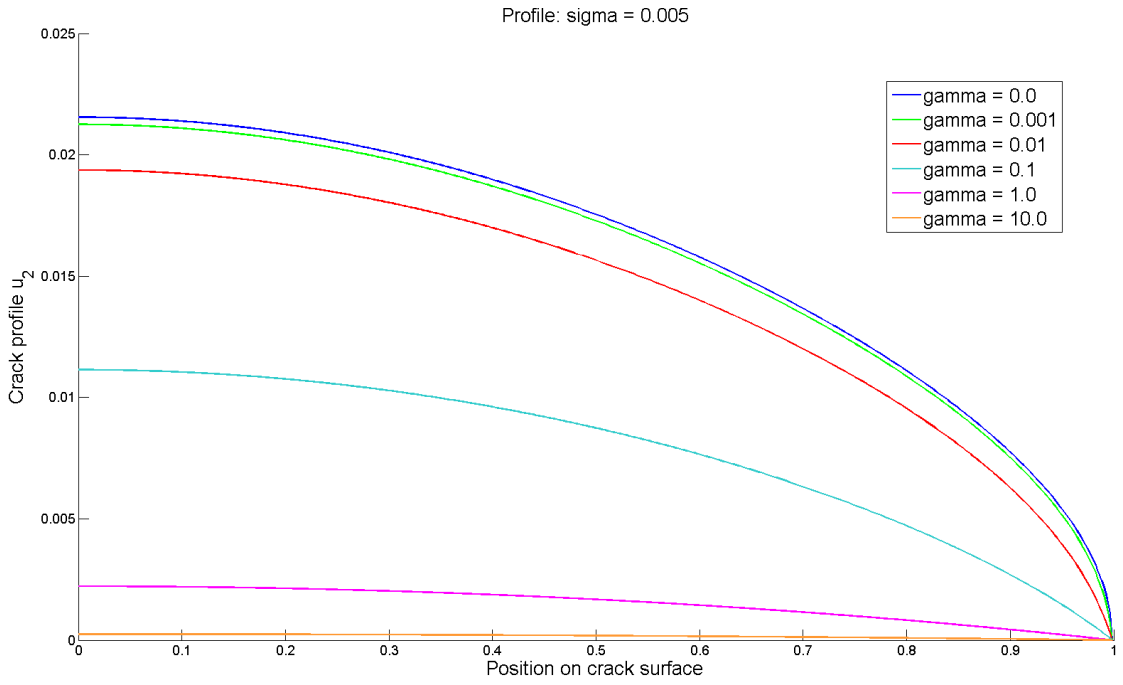


Figure 3.6 Crack profiles for $\sigma = 0.005$

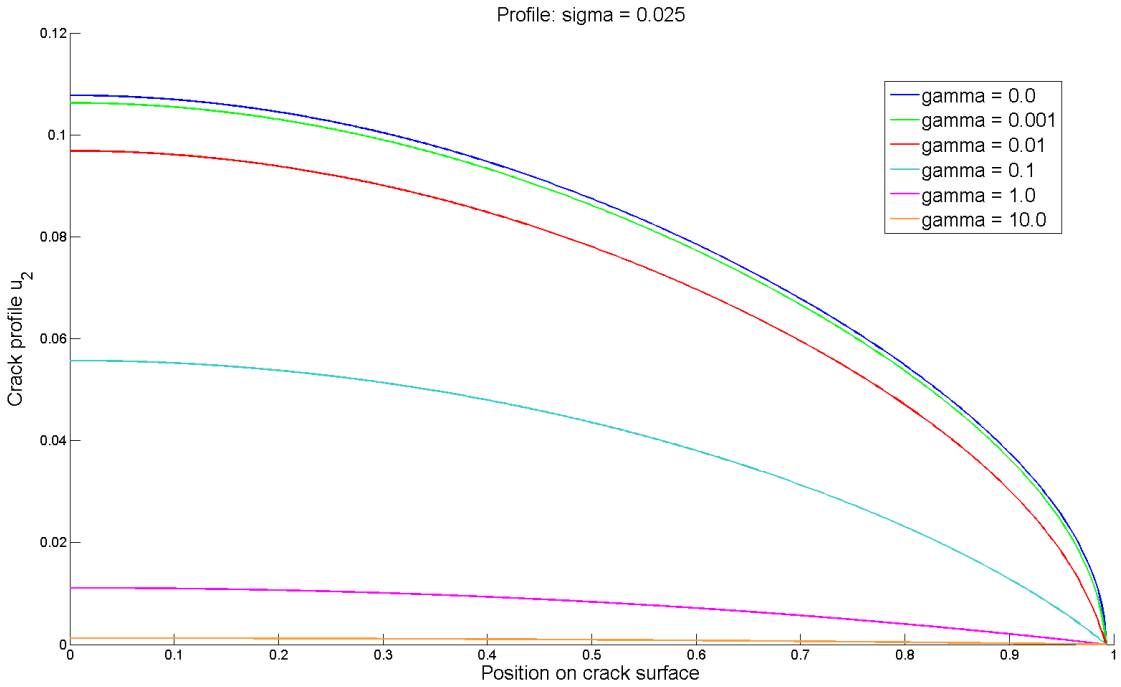


Figure 3.7 Crack profiles for $\sigma = 0.025$

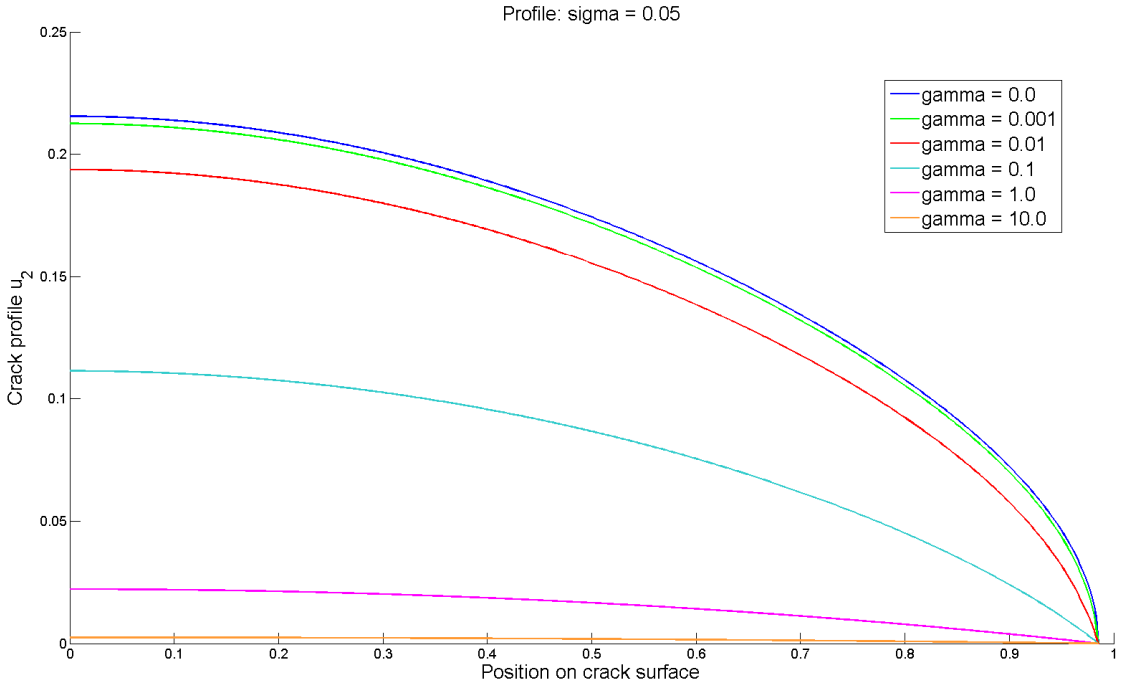
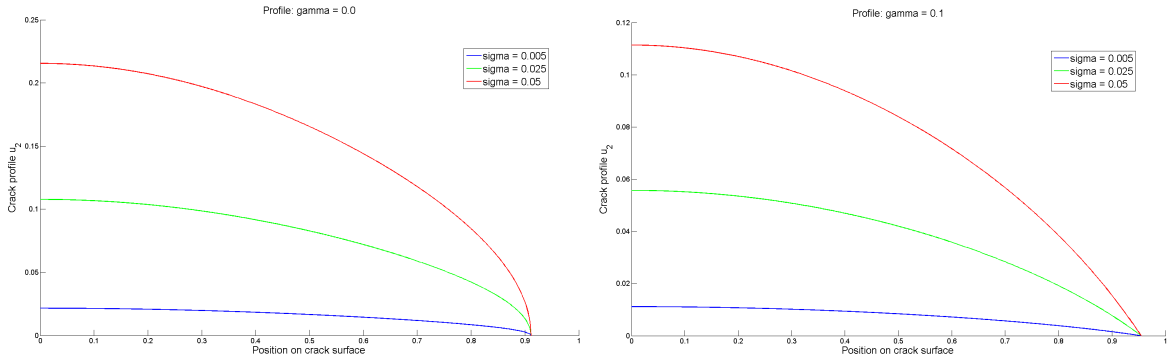
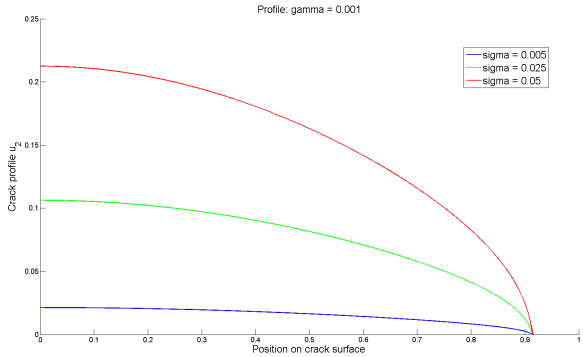


Figure 3.8 Crack profiles for $\sigma = 0.05$

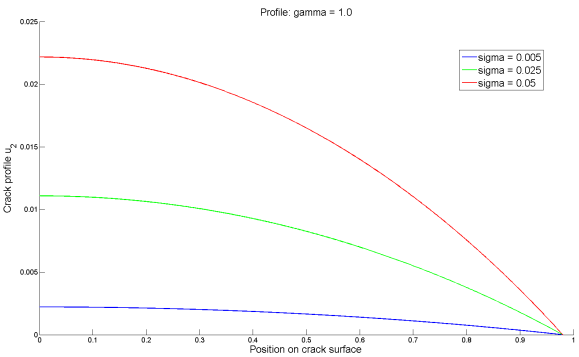


(a) $\gamma = 0.0$

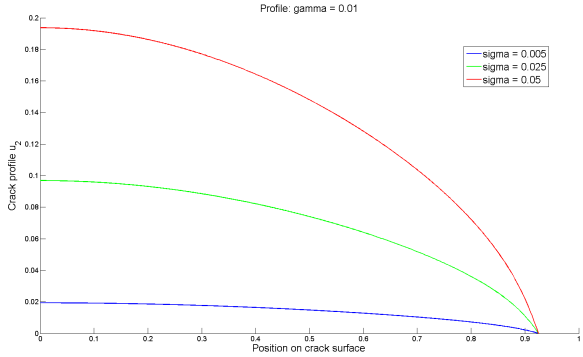
(b) $\gamma = 0.1$



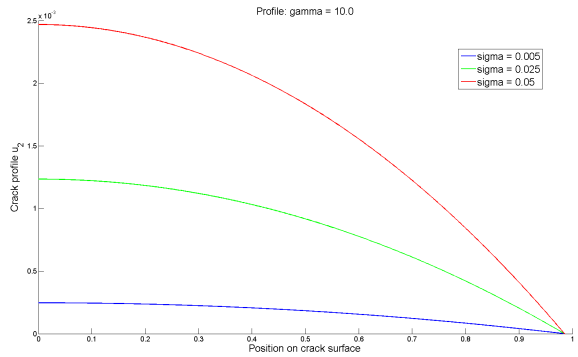
(c) $\gamma = 0.001$



(d) $\gamma = 1.0$



(e) $\gamma = 0.01$



(f) $\gamma = 10.0$

Figure 3.9 Crack profiles for various γ for increasing values of σ

the constant surface tension γ_0. As expected, we see that the deformation increases with the loading. These and the previous profile figures appear to conform to one of the predictions made by the Sendova-Walton model, which is that the profile exhibits a finite opening angle at the crack-tip for nonzero constant surface tension. For zero surface tension, we recover the LEFM model, which predicts an elliptical profile at the tip, as shown in the figures.

Even for a small nonzero constant surface tension, we still obtain a finite opening angle. This may not be obvious from the figures, but Tables 3.6 – 3.8 show the convergence of the opening angle at the crack tip for both zero and various nonzero values of surface tension, for $\sigma = 0.005$, 0.025, and 0.05, respectively. Note that this is not a perfect measurement of the angle, since it is computed as the angle that the cell immediately to the left of the tip makes with the X_1-axis after deformation, and is therefore limited by the structure of the mesh. For each value of σ, we see that the angle corresponding to zero surface tension is increasing to a large angle, approximately 80°. Since we measure the angle from the mesh, we cannot hope to obtain a perfect 90° because the deformed cell producing such an angle would be degenerate, and hence disallowed by the mesh deformation. However, we see that even a small nonzero surface tension produces a finite opening angle. As with the total deformation, this angle decreases as the constant surface tension increases.

Since the measurement of the opening angle from the deformed mesh is not very precise, we consider the slope of the crack profile as further evidence for a finite opening angle. Figures 3.10 – 3.12 show the slope $u_{2,1}$ of the crack profile, computed directly from the solution, for $\sigma = 0.005$, 0.025, and 0.05, respectively, as the constant surface tension increases. We see that the near-tip slope in the case of zero surface tension is significantly higher (in absolute value) than the slopes for nonzero surface tension. This is again consistent with the prediction of a finite opening angle for nonzero surface

Table 3.6 Convergence of the opening angle (°) across refinement cycles for $\sigma = 0.005$

Cycle	$\gamma_0 = 0.0$	$\gamma_0 = 0.001$	$\gamma_0 = 0.01$	$\gamma_0 = 0.1$	$\gamma_0 = 1.0$	$\gamma_0 = 10.0$
0	0.672694	0.669554	0.642562	0.457944	0.118233	0.0140449
1	1.42366	1.41415	1.33407	0.853647	0.186609	0.0211918
2	2.2183	2.19731	2.02601	1.15563	0.223987	0.0248015
3	3.35248	3.30238	2.92001	1.4253	0.245632	0.0266401
4	4.82121	4.70104	3.88907	1.61457	0.257647	0.0275718
5	6.86627	6.56716	4.90962	1.7442	0.263926	0.0280402
6	9.69481	8.93598	5.85512	1.82906	0.267353	0.0282775
7	13.4792	11.7133	6.59933	1.87908	0.269159	0.0283969
8	18.4851	14.5976	7.10248	1.90657	0.270082	0.028457
9	24.8782	17.229	7.41393	1.91987	0.270523	0.0284867
10	32.489	19.194	7.57664	1.92512	0.270723	0.0285013
11	40.8572	20.4174	7.63819	1.92583	0.270762	0.028508
12	49.1922	20.9981	7.63822	1.92383	0.270754	0.028511
13	56.7125	21.117	7.6059	1.92056	0.270686	0.0285118
14	62.9637	20.9489	7.55311	1.91655	0.27062	0.0285118
15	67.8573	20.6196	7.48937	1.91228	0.270528	0.0285112
16	71.5397	20.2074	7.4225	1.90734	0.270436	0.0285104
17	74.2448	19.7587	7.35346	1.90325	0.270338	0.0285103
18	76.2026	19.3013	7.28394	1.89816	0.270242	0.0285094
19	77.6076	18.8476	7.21113	1.89328	0.27025	0.0285084
20	78.6103	18.4078	7.14156	1.8885	0.27014	0.0285072

Table 3.7 Convergence of the opening angle (°) across refinement cycles for $\sigma = 0.025$

Cycle	$\gamma_0 = 0.0$	$\gamma_0 = 0.001$	$\gamma_0 = 0.01$	$\gamma_0 = 0.1$	$\gamma_0 = 1.0$	$\gamma_0 = 10.0$
0	3.40631	3.39031	3.25287	2.31431	0.59538	0.0706397
1	7.14406	7.09666	6.6971	4.29182	0.938618	0.106571
2	10.9417	10.8407	10.0144	5.76113	1.12474	0.1247
3	16.0759	15.8482	14.0964	7.03199	1.23128	0.133919
4	22.1769	21.6743	18.2085	7.8811	1.28931	0.138577
5	29.6431	28.5442	22.1361	8.4199	1.31858	0.140907
6	38.1185	35.7881	25.306	8.73551	1.33361	0.142076
7	46.6044	42.3944	27.3497	8.88054	1.34059	0.142653
8	54.4855	47.411	28.3018	8.91792	1.34318	0.142932
9	61.1968	50.5327	28.5143	8.89076	1.34339	0.143059
10	66.5074	51.751	28.2449	8.82954	1.34259	0.143112
11	70.5339	51.5562	27.676	8.75018	1.3406	0.143121
12	73.5156	50.4344	26.9573	8.66051	1.33886	0.143117
13	75.6764	48.7741	26.199	8.56808	1.33636	0.143098
14	77.2312	46.8618	25.4215	8.47553	1.33436	0.143075
15	78.342	44.8755	24.6489	8.38585	1.3319	0.14305
16	79.1327	42.9119	23.9229	8.28848	1.32955	0.143024
17	79.6947	41.0208	23.2271	8.21029	1.32713	0.143024
18	80.0931	39.2328	22.5645	8.11595	1.32477	0.142997
19	80.3756	37.5495	21.9024	8.02742	1.32496	0.142972
20	80.5756	35.9828	21.2947	7.94252	1.32232	0.142941

Table 3.8 Convergence of the opening angle (°) across refinement cycles for $\sigma = 0.05$

Cycle	$\gamma_0 = 0.0$	$\gamma_0 = 0.001$	$\gamma_0 = 0.01$	$\gamma_0 = 0.1$	$\gamma_0 = 1.0$	$\gamma_0 = 10.0$
0	6.90745	6.87493	6.59538	4.68684	1.20139	0.142331
1	14.2202	14.1283	13.3517	8.61383	1.89104	0.214692
2	21.0948	20.9131	19.4148	11.4121	2.26109	0.251154
3	29.4663	29.1004	26.2322	13.7184	2.46969	0.269657
4	38.0444	37.3433	32.3096	15.1534	2.58055	0.278972
5	46.8071	45.5089	37.3493	15.9656	2.63368	0.283599
6	54.9259	52.5635	40.7545	16.3566	2.65851	0.285893
7	61.5335	57.8146	42.415	16.4351	2.66742	0.286996
8	66.7819	61.0079	42.6396	16.3244	2.66762	0.287499
9	70.7611	62.3808	42.012	16.1086	2.66321	0.287697
10	73.6854	62.1761	40.9003	15.8445	2.65729	0.287753
11	75.796	60.8707	39.5046	15.5587	2.64806	0.287708
12	77.3207	58.8733	38.0005	15.2634	2.64055	0.28765
13	78.4049	56.4913	36.5227	14.9728	2.63044	0.287553
14	79.178	53.9582	35.0739	14.6915	2.62248	0.287451
15	79.7267	51.421	33.675	14.4251	2.61289	0.287345
16	80.1157	48.9593	32.388	14.1418	2.60381	0.287237
17	80.3916	46.6129	31.1755	13.9179	2.59453	0.287236
18	80.5868	44.4086	30.0378	13.6519	2.58551	0.287127
19	80.7251	42.3432	28.9166	13.406	2.58625	0.287024
20	80.8229	40.4284	27.9009	13.1734	2.57622	0.286899

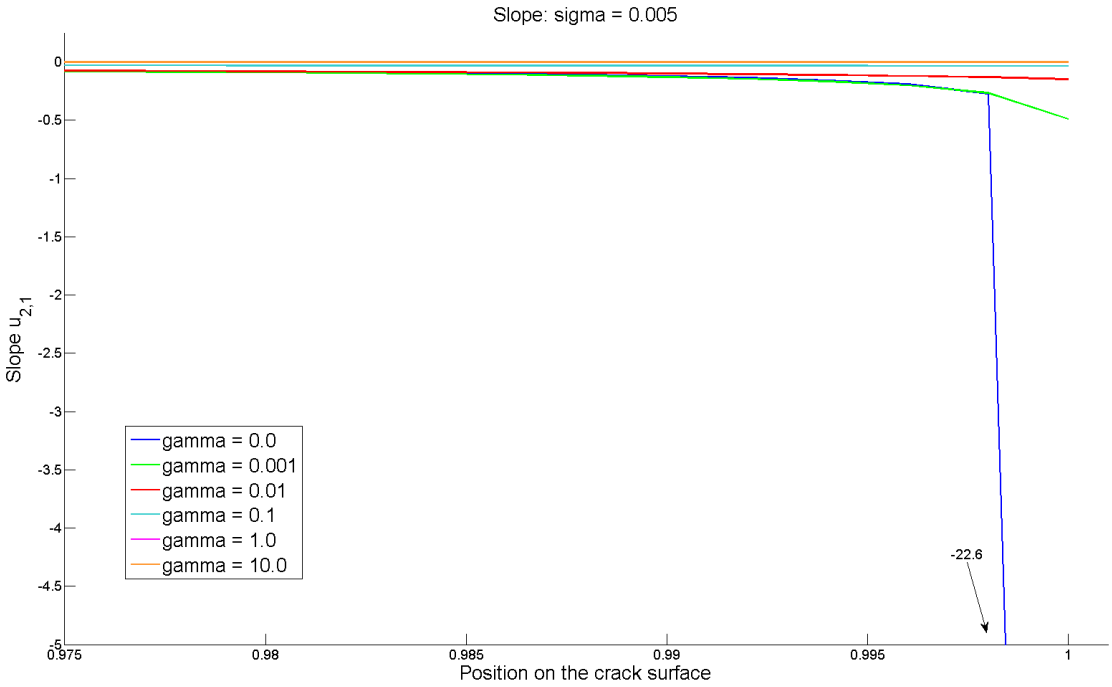


Figure 3.10 Near-tip slope $u_{2,1}$ of the crack profile for $\sigma = 0.005$

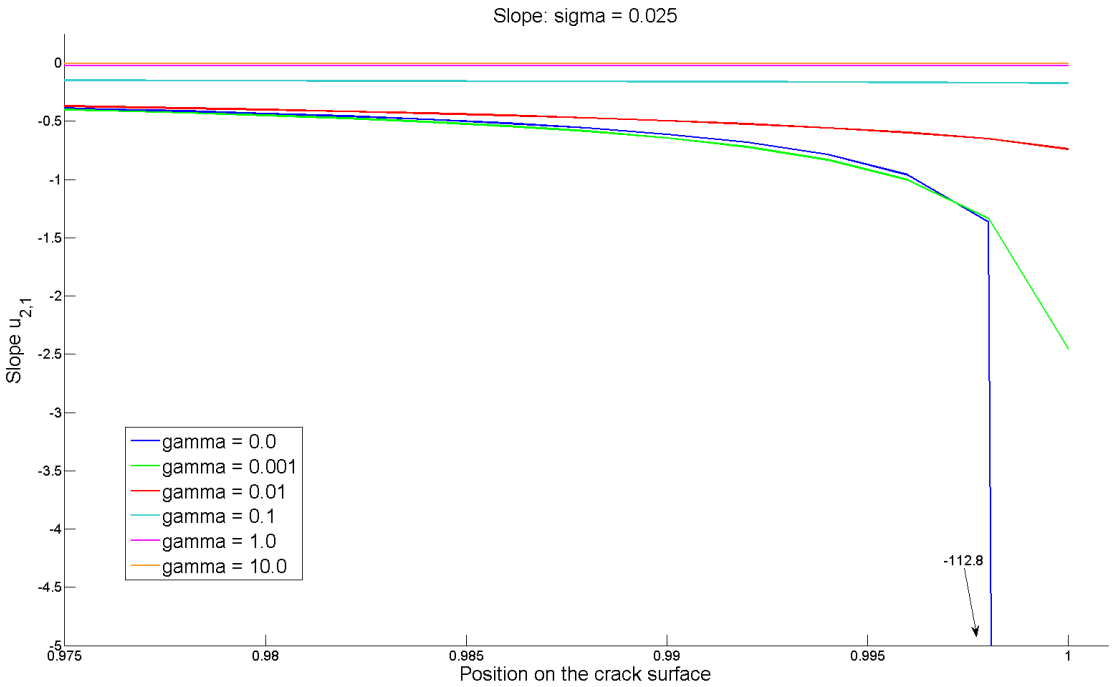


Figure 3.11 Near-tip slope $u_{2,1}$ of the crack profile for $\sigma = 0.025$

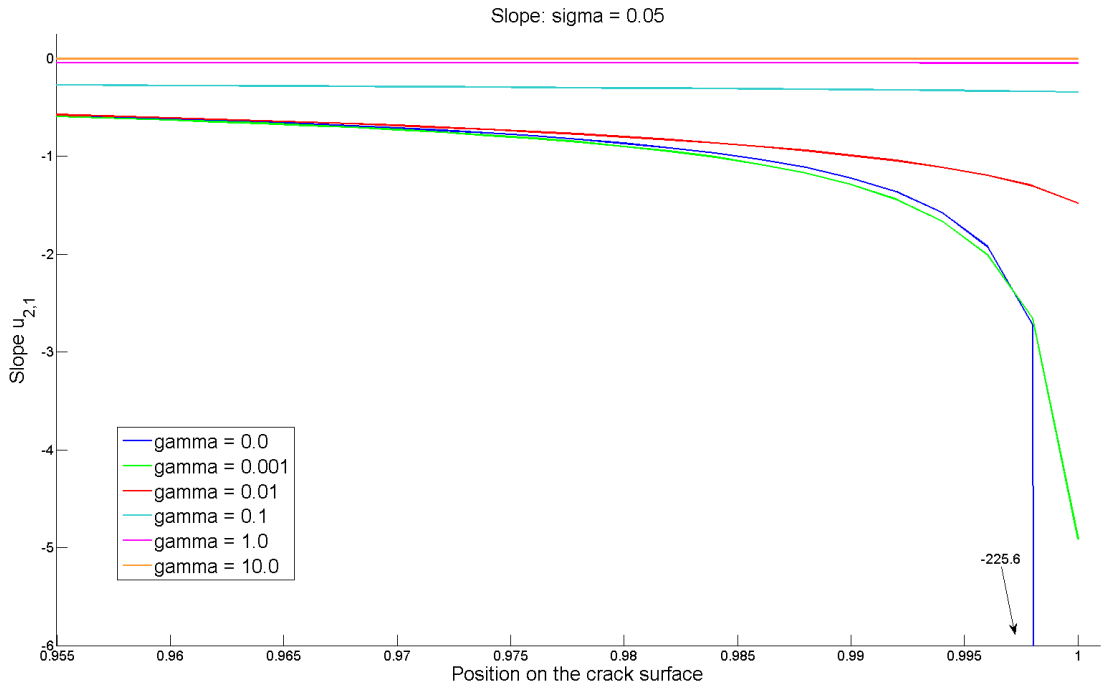


Figure 3.12 Near-tip slope $u_{2,1}$ of the crack profile for $\sigma = 0.05$

tension. For the zero surface tension case, LEFM predicts an infinite slope, which of course we cannot attain, but is indicated by the high values of the slope. These values will only increase as one approaches even closer to the crack tip.

The other important trend predicted by the Sendova-Walton model is that the crack-tip stress singularity is reduced from a square root to a logarithmic singularity. We certainly see strong evidence for this from our results. We show the stress along the bottom face leading up to the crack for increasing values of γ_0 and $\sigma = 0.005$, 0.025, and 0.05, respectively in Figures 3.13 – 3.15. The nonzero surface tension cases show near tip stresses that are significantly less than that of the zero surface tension case, even for very small nonzero values. Although we have not yet conducted any rigorous computations to prove that this reduction in the near-tip stress behavior corresponds to a logarithmic singularity, by simply comparing the profiles visually with the two

functions $1/\sqrt{r}$ and $-\log(r)$, where r is the distance to the crack-tip, we see that the zero surface tension cases behave similarly to the square root function, while the nonzero cases are closer to the log function. Figure 3.16 shows six sets of stress data, one for zero surface tension and five for various values of nonzero surface tension. Each set has three stress graphs, corresponding to the value of the loading σ, along with graphs of the square root and log functions.

Together, these results show that the numerical implementation of the Sendova-Walton model applied to the classical Griffith crack problem in the case of constant surface tension strongly agrees with the predicted behavior of the Sendova-Walton model. In particular, we show that for a nonzero constant surface tension, the crack-tip opening angle is finite, with a corresponding finite slope, and the crack-tip stress singularity is reduced to a logarithmic singularity. In addition, we see two other expected trends in the data, namely that the deformation increases with the far-field loading and decreases as the constant surface tension increases.

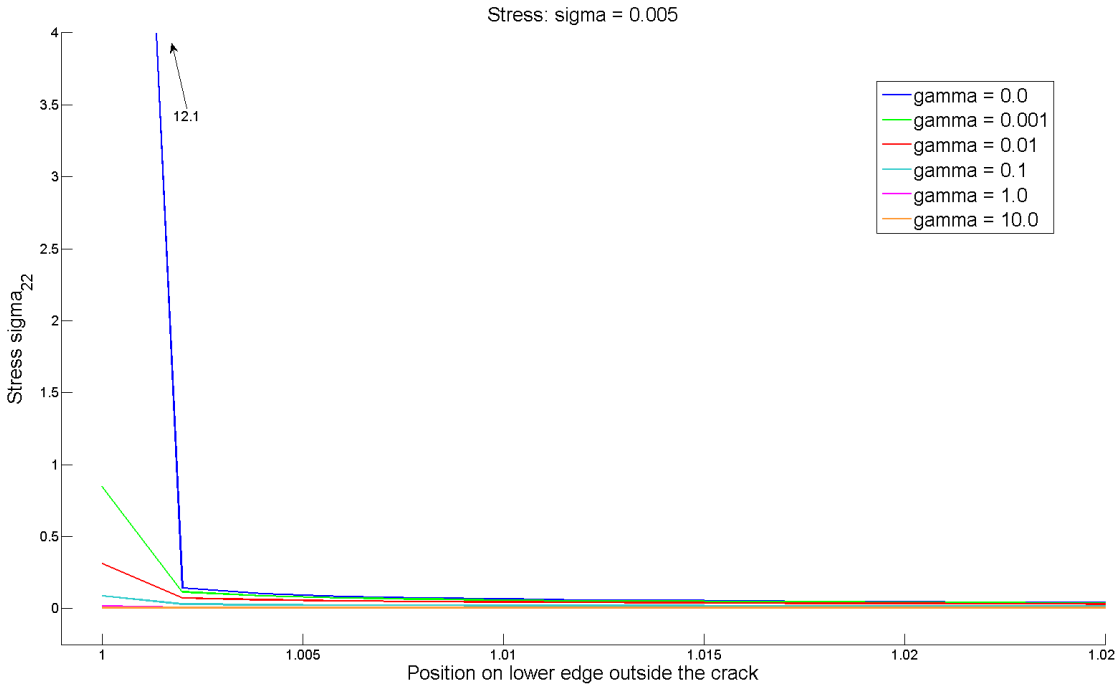


Figure 3.13 Near-tip stress σ_{22} outside the crack for $\sigma = 0.005$

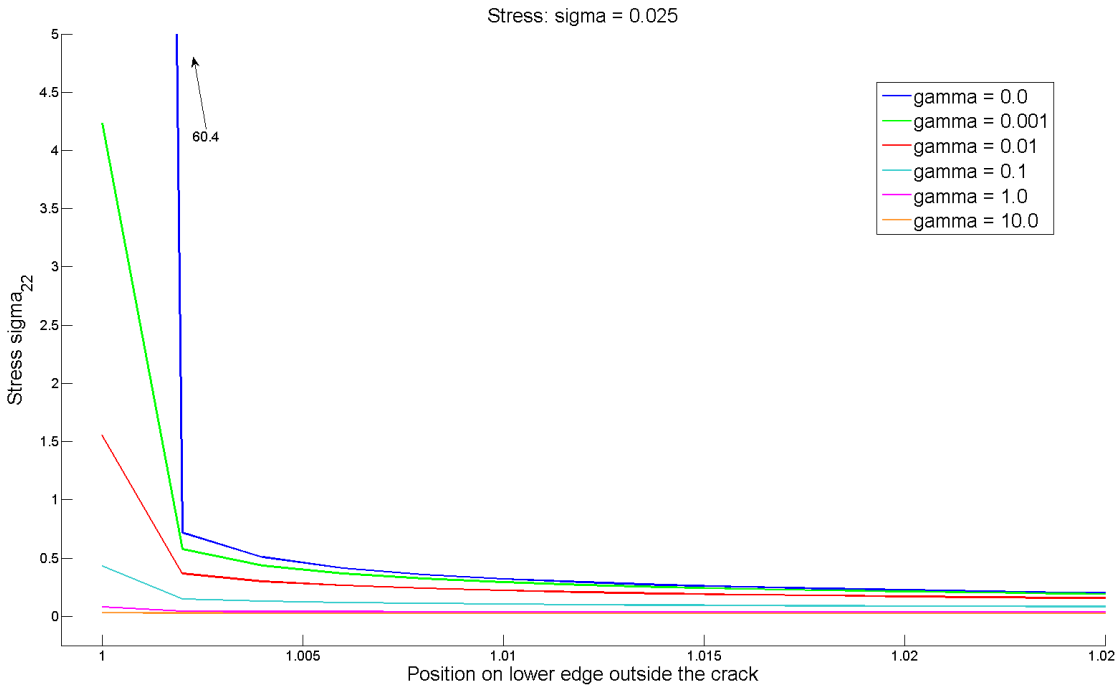


Figure 3.14 Near-tip stress σ_{22} outside the crack for $\sigma = 0.025$

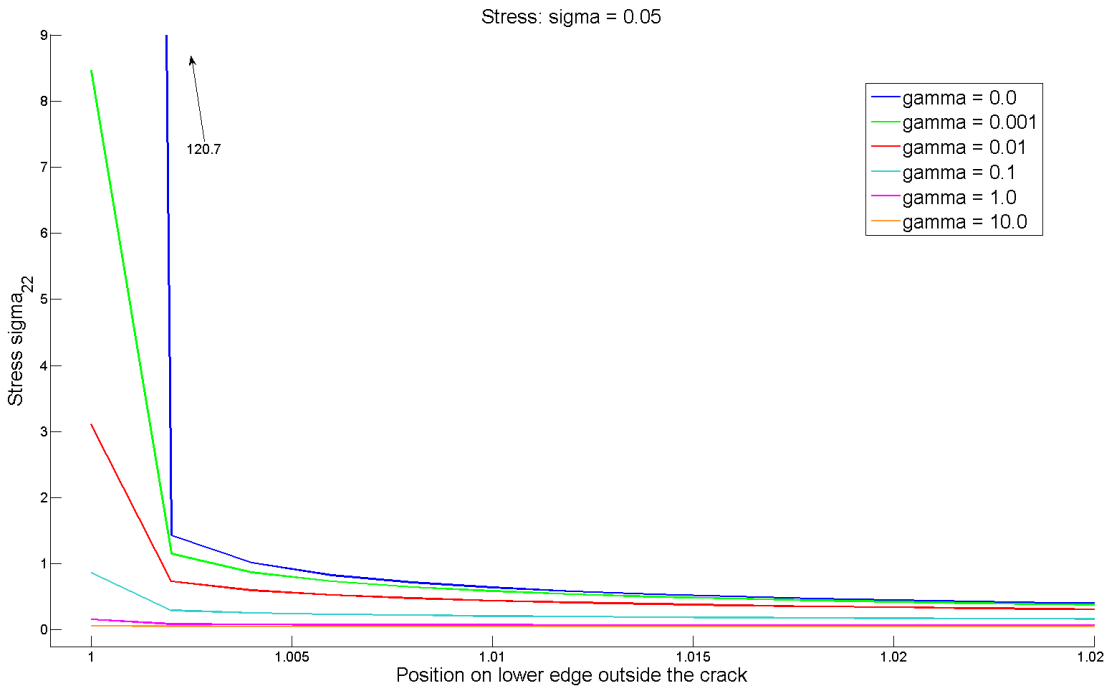
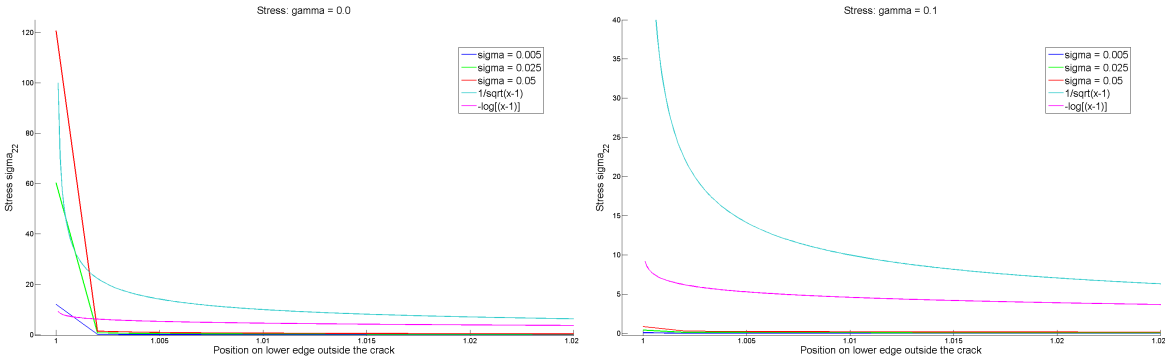
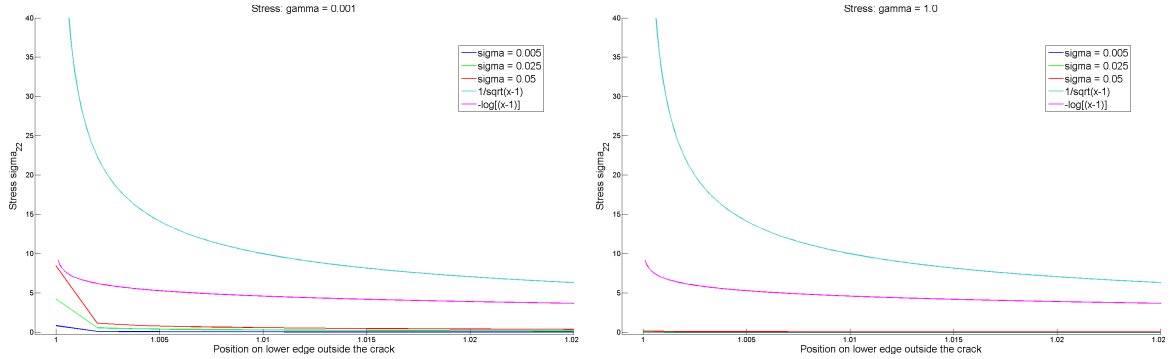


Figure 3.15 Near-tip stress σ_{22} outside the crack for $\sigma = 0.05$



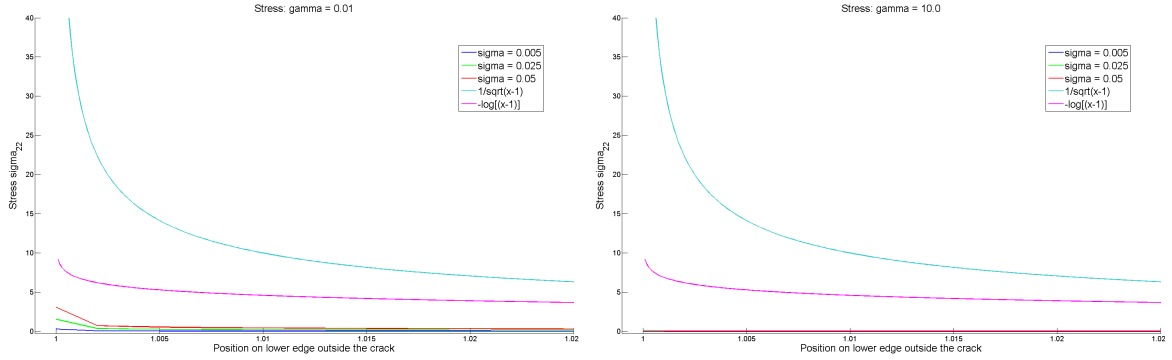
(a) $\gamma = 0.0$

(b) $\gamma = 0.1$



(c) $\gamma = 0.001$

(d) $\gamma = 1.0$



(e) $\gamma = 0.01$

(f) $\gamma = 10.0$

Figure 3.16 Near-tip stress σ_{22} outside the crack for various γ for increasing values of σ

4. DETERMINATION OF THE CURVATURE-DEPENDENT SURFACE TENSION PARAMETERS

One of the major steps required to implement the Sendova-Walton fracture theory in the case of curvature-dependent surface tension is to identify the appropriate ranges of the surface tension parameters which guarantee bounded crack-tip stresses and a cusp-like opening profile. In this section, we present a new proof of the existence of such parameters and construct an algorithm for determining their ranges. We note that the theory developed by Sendova and Walton considers the entire upper crack surface, i.e., using symmetry to consider the problem on just the upper half-plane. They did not further reduce to the upper right quarter-plane, and neither do we for the remainder of this section. This theory also applies to the infinite half-plane, not an approximating finite computational domain as in Section 3. In particular, we note that the entire (upper) crack surface in the reference configuration is given by

$$\Sigma^+ = (x, 0^+), \quad \text{for} \quad -1 \leq x \leq 1. \tag{4.1}$$

(Note that, for simplicity, we have changed notation from (X_1, X_2) to (x, y) and we will drop the reference subscript κ.)

The (linearized) curvature-dependent surface tension that we consider is given by Sendova and Walton (2010) as

$$\tilde{\gamma}(x) = \gamma_0 + \gamma_1 u_{2,11}(x, 0) + \text{h.o.t.}, \tag{4.2}$$

where the parameters γ_0 and γ_1 are both nondimensional constants. As in the constant surface tension case, we apply this surface tension to the component-form JMB

equations (2.6) and (2.7) and linearize by assuming $u_{j,k}$ and $u_{i,jk}$ are small whenever $j \neq k$. This yields the linearized JMB equations for the curvature-dependent case

$$\left.\begin{aligned}\sigma_{12}(x,0) &= -\gamma_1 u_{2,111}(x,0) + \text{h.o.t.} \\ \sigma_{22}(x,0) &= -\gamma_0 u_{2,11}(x,0) + \text{h.o.t.}\end{aligned}\right\}, \text{ for } |x| \leq 1. \tag{4.3}$$

(Note the sign correction, cf. Sendova and Walton (2010), equation (26).)

With this JMB condition, we will first prove the existence of parameters γ_0 and γ_1 that result in bounded stresses at the crack tip. We will then discuss the algorithm we developed to estimate threshold values for these parameters. The results of this algorithm are presented in Section 4.4.

4.1 Existence of Bounding Surface Tension Parameters

Sendova and Walton (2010) show, using Hooke's law, Fourier transforms, Dirichlet-to-Neumann and Neumann-to-Dirichlet maps, and the linearized boundary conditions in (4.3), that the differential momentum balance equation (with zero body force) may be transformed into a linear integro-differential equation

$$\gamma_0 \phi'(x) + \frac{1}{\pi} \fint_{-1}^{1} \frac{\zeta_1 \phi(r) - \zeta_2 \gamma_1 \phi''(r)}{r - x}\, dr = -\sigma, \quad x \in (-1,1), \tag{4.4}$$

where $\phi(x) = u_{2,1}(x, 0^+)$ (cf. Sendova and Walton 2010, equation (27)). Due to nondimensionalization, the constants ζ_1 and ζ_2 depend only on Poisson's ratio (ν):

$$\zeta_1 := \frac{1}{2(1-\nu^2)}, \qquad \zeta_2 := \frac{1-2\nu}{2(1-\nu)}. \tag{4.5}$$

They make two further assumptions which we adopt:

1. First, the crack opening profile, $u_2(x,0)$, should be symmetric across the y-axis, so it must be an even function of x, and therefore we assume that ϕ is odd, i.e.,

$$\phi(x) = -\phi(-x), \quad |x| < 1. \tag{4.6}$$

2. Second, since the crack surfaces at the tips should form a cusp, we require the slope of the opening profile, $u_{2,1}(x,0)$, to be continuous at the crack tips, i.e.,

$$\phi(\pm 1) = 0. \tag{4.7}$$

With these assumptions, Sendova and Walton (2010, Theorem 1.) prove that equation (4.4) has a unique solution for all but countably many values of γ_0 and γ_1. However, they do not give any insight into the actual ranges of values that these surface tension parameters must take. We present a slightly modified theorem and a new proof that we will use to construct an algorithm to find the threshold value of γ_1 for any given value of γ_0.

Theorem 4.1 *For each fixed value of γ_0, the singular linear integro-differential equation (4.4), subject to conditions (4.6) and (4.7), has a unique continuous (and hence non-singular) solution on $[-1,1]$ for all but countably many values of γ_1.*

The proof of this theorem follows from a series of lemmas.

Lemma 4.2 *The integro-differential equation (4.4), subject to conditions (4.6) and (4.7), may be reduced to canonical form by recasting it as a singular integral equation for $\psi(x)$, where*

$$\psi(x) := \phi''(x) = u_{2,111}(x, 0^+). \tag{4.8}$$

Proof. First, we define two auxiliary functions

$$\kappa(x) := x\left(1 - \log|x|\right), \tag{4.9}$$

$$\lambda(x) := \kappa(1-x) + \kappa(1+x). \tag{4.10}$$

We claim that

$$⨍_{-1}^{1} \frac{\phi(r)}{r-x}\,dr = \phi'(1)\lambda(x) - ⨍_{-1}^{1} \psi(r)\kappa(r-x)\,dr. \tag{4.11}$$

To see this, we note that the Cauchy principal value on the left-hand side is defined by

$$⨍_{-1}^{1} \frac{\phi(r)}{r-x}\,dr := \lim_{\varepsilon\to 0^+}\left[\int_{-1}^{x-\varepsilon} \frac{\phi(r)}{r-x}\,dr + \int_{x+\varepsilon}^{1} \frac{\phi(r)}{r-x}\,dr\right]. \tag{4.12}$$

We can evaluate this limit after integrating by parts and applying condition (4.7), which yields

$$\begin{aligned}
&⨍_{-1}^{1} \frac{\phi(r)}{r-x}\,dr \\
&\quad= \lim_{\varepsilon\to 0^+}\left[\log|r-x|\phi(r)\Big|_{-1}^{x-\varepsilon} + \log|r-x|\phi(r)\Big|_{x+\varepsilon}^{1} - \int_{-1}^{1}\log|r-x|\phi'(r)\,dr\right] \\
&\quad= -⨍_{-1}^{1}\log|r-x|\phi'(r)\,dr \\
&\quad= -(r-x)\left[\log|r-x|-1\right]\phi'(r)\Big|_{-1}^{1} + ⨍_{-1}^{1}(r-x)\left[\log|r-x|-1\right]\phi''(r)\,dr \\
&\quad= \phi'(1)\lambda(x) - ⨍_{-1}^{1}\psi(r)\kappa(r-x)\,dr.
\end{aligned} \tag{4.13}$$

We apply this result to (4.4) and note that

$$\phi'(x) = \int_{-1}^{x}\psi(r)\,dr + \phi'(1). \tag{4.14}$$

After rearranging terms, this yields the canonical form

$$-\frac{\gamma_1\zeta_2}{\pi}⨍_{-1}^{1}\frac{\psi(r)}{r-x}\,dr = -\sigma-\phi'(1)\left[\gamma_0+\frac{\zeta_1}{\pi}\lambda(x)\right]-\gamma_0\int_{-1}^{x}\psi(r)\,dr+\frac{\zeta_1}{\pi}⨍_{-1}^{1}\psi(r)\kappa(r-x)\,dr, \tag{4.15}$$

which is a singular integral equation for $\psi(x)$. □

Lemma 4.3 *If $\psi(x)$ solves (4.15), then*

$$\begin{aligned}0 = &-(\sigma+\gamma_0\phi'(1))\pi-\phi'(1)\frac{\zeta_1}{\pi}⨍_{-1}^{1}\frac{\lambda(x)}{\sqrt{1-x^2}}\,dx+\gamma_0\int_{-1}^{1}\psi(r)\sin^{-1}(r)\,dr\\ &+\frac{\zeta_1}{\pi}\int_{-1}^{1}\frac{dx}{\sqrt{1-x^2}}⨍_{-1}^{1}\psi(r)\kappa(r-x)\,dr.\end{aligned} \tag{4.16}$$

Proof. We will multiply equation (4.15) by $\frac{1}{\sqrt{1-x^2}}$ and integrate with respect to x over the interval $(-1,1)$. Note that the function $\frac{1}{\sqrt{1-x^2}}$ spans the null space of the finite Hilbert transform (Jiang and Rokhlin 2003), which is defined by the Cauchy principal value

$$\mathcal{H}[g](x) = \frac{1}{\pi}⨍_{-1}^{1}g(r)\frac{dr}{r-x}. \tag{4.17}$$

In other words,

$$⨍_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)} = 0,\ x\in(-1,1). \tag{4.18}$$

We will make frequent use of this fact. As a first instance, we note that the first term of (4.15) will vanish since changing the order of integration yields

$$\int_{-1}^{1}\frac{dx}{\sqrt{1-x^2}}⨍_{-1}^{1}\frac{\psi(r)}{r-x}\,dr = -\int_{-1}^{1}\psi(r)\,dr⨍_{-1}^{1}\frac{dx}{\sqrt{1-x^2}(x-r)}. \tag{4.19}$$

Thus equation (4.15) becomes

$$0 = -(\sigma + \gamma_0\phi'(1)) \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} - \phi'(1)\frac{\zeta_1}{\pi} \fint_{-1}^{1} \frac{\lambda(x)}{\sqrt{1-x^2}}\, dx$$
$$- \gamma_0 \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} \int_{-1}^{x} \psi(r)\, dr + \frac{\zeta_1}{\pi} \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} \fint_{-1}^{1} \psi(r)\kappa(r-x)\, dr. \qquad (4.20)$$

Note that

$$\int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} = \sin^{-1}(x)\Big|_{-1}^{1} = \pi. \qquad (4.21)$$

We also change the order of integration in the third term on the right-hand side of (4.20) which becomes

$$\begin{aligned} -\gamma_0 \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} \int_{-1}^{x} \psi(r)\, dr &= -\gamma_0 \int_{-1}^{1} \psi(r)\, dr \int_{r}^{1} \frac{dx}{\sqrt{1-x^2}} \\ &= -\gamma_0 \int_{-1}^{1} \psi(r) \left[\frac{\pi}{2} - \sin^{-1}(r)\right] dr \\ &= \gamma_0 \int_{-1}^{1} \psi(r) \sin^{-1}(r)\, dr, \end{aligned} \qquad (4.22)$$

since $\psi(r) = \phi''(r)$ is odd.

Substituting these results back into (4.20) yields

$$0 = -(\sigma + \gamma_0\phi'(1))\pi - \phi'(1)\frac{\zeta_1}{\pi} \fint_{-1}^{1} \frac{\lambda(x)}{\sqrt{1-x^2}}\, dx$$
$$+ \gamma_0 \int_{-1}^{1} \psi(r) \sin^{-1}(r)\, dr + \frac{\zeta_1}{\pi} \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} \fint_{-1}^{1} \psi(r)\kappa(r-x)\, dr. \qquad (4.23)$$

□

Lemma 4.4 *The singular integral equation (4.15), subject to conditions (4.6) and (4.7), is equivalent to the equation*

$$\gamma_1\zeta_2\psi(x) = \sqrt{1-x^2}\left[-\phi'(1)\frac{\zeta_1}{\pi^2}⨍_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right. \tag{4.24}$$
$$\left.-\frac{\gamma_0}{\pi}⨍_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)}\int_{-1}^{r}\psi(q)\,dq + \frac{\zeta_1}{\pi^2}\int_{-1}^{1}\psi(q)\,dq⨍_{-1}^{1}\frac{\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right].$$

Proof. First, we observe that since $\psi(x)$ is odd, so is $\kappa(x)$, while $\lambda(x)$ is even. We then apply the operator

$$\mathcal{K}[f] := \frac{1}{\pi}⨍_{-1}^{1} f(r)\sqrt{1-r^2}\frac{dr}{r-x} \tag{4.25}$$

to both sides of (4.15). We will look at each resulting term individually.

In the following, we will make use of the Poincaré-Bertrand formula (Muskhelishvili 1977, §23) to switch the order of integration of a double Cauchy principle value, which is given by

$$⨍_{-1}^{1}\frac{dr}{r-x}⨍_{-1}^{1} f(r,q)\frac{dq}{q-r} = -\pi^2 f(x,x) + ⨍_{-1}^{1}dq⨍_{-1}^{1} f(r,q)\frac{dr}{(q-r)(r-x)}. \tag{4.26}$$

For the term on the left-hand side of (4.15), we apply $\mathcal{K}$ and use the Poincaré-Bertrand formula to obtain

$$\mathcal{K}\left[-\frac{\gamma_1\zeta_2}{\pi}⨍_{-1}^{1}\frac{\psi(r)}{r-x}\,dr\right] = -\frac{\gamma_1\zeta_2}{\pi^2}⨍_{-1}^{1}\left(⨍_{-1}^{1}\frac{\psi(q)}{q-r}\,dq\right)\sqrt{1-r^2}\frac{dr}{r-x}$$
$$= -\frac{\gamma_1\zeta_2}{\pi^2}\left[-\pi^2\psi(x)\sqrt{1-x^2} + ⨍_{-1}^{1}dq⨍_{-1}^{1}\psi(q)\sqrt{1-r^2}\frac{dr}{(q-r)(r-x)}\right]. \tag{4.27}$$

It turns out that the integral on the right-hand side above is identically zero. To see this, we first compute the integral $⨍_{-1}^{1}\sqrt{1-r^2}/(r-x)\,dr$. We use a standard trick of adding and subtracting a term in the numerator to cancel a singularity in the

denominator. This yields

$$\begin{aligned}
\fint_{-1}^{1} \sqrt{1-r^2}\frac{dr}{r-x} &= \fint_{-1}^{1} \frac{1-r^2}{\sqrt{1-r^2}}\frac{dr}{r-x} \\
&= \fint_{-1}^{1} \frac{(1-r^2)-(1-x^2)}{\sqrt{1-r^2}}\frac{dr}{r-x} + (1-x^2)\fint_{-1}^{1} \frac{dr}{\sqrt{1-r^2}(r-x)} \\
&= -\fint_{-1}^{1} \frac{r+x}{\sqrt{1-r^2}}\,dr + 0 \\
&= -\fint_{-1}^{1} \frac{r}{\sqrt{1-r^2}}\,dr - x\fint_{-1}^{1} \frac{dr}{\sqrt{1-r^2}} \\
&= 0 - \pi x, \qquad (4.28)
\end{aligned}$$

where we have used the Hilbert transform null space result from (4.18) and the fact that $\frac{r}{\sqrt{1-r^2}}$ is odd.

Using this computation, we find that the integral on the right-hand side of (4.27) becomes

$$\begin{aligned}
&\fint_{-1}^{1} \psi(q)\,dq \fint_{-1}^{1} \frac{\sqrt{1-r^2}}{(q-r)(r-x)}\,dr \\
&\quad= \fint_{-1}^{1} \psi(q)\,dq \fint_{-1}^{1} \sqrt{1-r^2}\left[\frac{1}{q-r} + \frac{1}{r-x}\right]\frac{dr}{q-x} \\
&\quad= \fint_{-1}^{1} \psi(q)\frac{dq}{q-x}\left[\fint_{-1}^{1} \sqrt{1-r^2}\frac{dr}{q-r} + \fint_{-1}^{1} \sqrt{1-r^2}\frac{dr}{r-x}\right] \\
&\quad= \fint_{-1}^{1} \psi(q)\frac{dq}{q-x}[\pi q - \pi x] \\
&\quad= \pi\fint_{-1}^{1} \psi(q)\,dq \\
&\quad= 0, \qquad (4.29)
\end{aligned}$$

since $\psi(q)$ is odd.

Thus (4.27) becomes

$$\mathcal{K}\left[-\frac{\gamma_1\zeta_2}{\pi}\fint_{-1}^{1}\frac{\psi(r)}{r-x}\,dr\right]=\gamma_1\zeta_2\psi(x)\sqrt{1-x^2}. \tag{4.30}$$

Next, we apply $\mathcal{K}$ to the constant term on the right-hand side of (4.15) and use the result in (4.28) to obtain

$$\begin{aligned}\mathcal{K}\left[-(\sigma+\gamma_0\phi'(1))\right]&=-(\sigma+\gamma_0\phi'(1))\frac{1}{\pi}\fint_{-1}^{1}\sqrt{1-r^2}\frac{dr}{r-x}\\&=-(\sigma+\gamma_0\phi'(1))\frac{1}{\pi}(-\pi x)\\&=(\sigma+\gamma_0\phi'(1))x.\end{aligned} \tag{4.31}$$

Applying $\mathcal{K}$ to the term involving $\lambda(x)$ on the right-hand side of (4.15), we use our standard trick to remove the singularity at $r=x$, which yields

$$\begin{aligned}&\mathcal{K}\left[-\phi'(1)\frac{\zeta_1}{\pi}\lambda(x)\right]\\&\quad=-\phi'(1)\frac{\zeta_1}{\pi^2}\fint_{-1}^{1}\lambda(r)\sqrt{1-r^2}\frac{dr}{r-x}\\&\quad=-\phi'(1)\frac{\zeta_1}{\pi^2}\left[\fint_{-1}^{1}\lambda(r)\frac{(1-r^2)-(1-x^2)}{\sqrt{1-r^2}}\frac{dr}{r-x}+(1-x^2)\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right]\\&\quad=-\phi'(1)\frac{\zeta_1}{\pi^2}\left[-\fint_{-1}^{1}\lambda(r)\frac{r+x}{\sqrt{1-r^2}}\,dr+(1-x^2)\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right]\\&\quad=-\phi'(1)\frac{\zeta_1}{\pi^2}\left[-x\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\,dr+(1-x^2)\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right],\end{aligned} \tag{4.32}$$

where we note that $\frac{\lambda(r)r}{\sqrt{1-r^2}}$ is odd since $\lambda(r)$ is even.

For the first integral term on the right-hand side of (4.15) we have

$$\begin{aligned}
&\mathcal{K}\left[-\gamma_0\int_{-1}^{x}\psi(r)\,dr\right]\\
&\quad=-\frac{\gamma_0}{\pi}\fint_{-1}^{1}\left(\int_{-1}^{r}\psi(q)\,dq\right)\sqrt{1-r^2}\,\frac{dr}{r-x}\\
&\quad=-\frac{\gamma_0}{\pi}\left[\fint_{-1}^{1}\left(\int_{-1}^{r}\psi(q)\,dq\right)\frac{(1-r^2)-(1-x^2)}{\sqrt{1-r^2}}\,\frac{dr}{r-x}\right.\\
&\qquad\left.+\,(1-x^2)\fint_{-1}^{1}\left(\int_{-1}^{r}\psi(q)\,dq\right)\frac{1}{\sqrt{1-r^2}}\,\frac{dr}{r-x}\right]\\
&\quad=-\frac{\gamma_0}{\pi}\left[-\fint_{-1}^{1}\frac{r+x}{\sqrt{1-r^2}}\,dr\int_{-1}^{r}\psi(q)\,dq\right.\\
&\qquad\left.+\,(1-x^2)\fint_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)}\fint_{-1}^{r}\psi(q)\,dq\right]\\
&\quad=-\frac{\gamma_0}{\pi}\left[-\fint_{-1}^{1}\frac{r(\phi'(r)-\phi'(1))}{\sqrt{1-r^2}}\,dr-x\fint_{-1}^{1}\frac{dr}{\sqrt{1-r^2}}\int_{-1}^{r}\psi(q)\,dq\right.\\
&\qquad\left.+\,(1-x^2)\fint_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)}\fint_{-1}^{r}\psi(q)\,dq\right].
\end{aligned}\tag{4.33}$$

The first integral on the above right-hand side vanishes since $\phi'(r)$ is even. We change the order of integration in the second integral to obtain

$$\begin{aligned}
&\mathcal{K}\left[-\gamma_0\int_{-1}^{x}\psi(r)\,dr\right]\\
&\quad=-\frac{\gamma_0}{\pi}\left[-x\fint_{-1}^{1}\psi(q)\,dq\int_{q}^{1}\frac{dr}{\sqrt{1-r^2}}+(1-x^2)\fint_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)}\fint_{-1}^{r}\psi(q)\,dq\right]\\
&\quad=-\frac{\gamma_0}{\pi}\left[x\fint_{-1}^{1}\psi(q)\sin^{-1}(q)\,dq+(1-x^2)\fint_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)}\fint_{-1}^{r}\psi(q)\,dq\right],
\end{aligned}\tag{4.34}$$

since $\psi(q)$ is odd.

Finally, we apply the operator $\mathcal{K}$ to the last integral on the right-hand side of (4.15) to obtain

$$\begin{aligned}
&\mathcal{K}\left[\frac{\zeta_1}{\pi}\fint_{-1}^{1}\psi(r)\kappa(r-x)\,dr\right] \\
&\quad=\frac{\zeta_1}{\pi^2}\fint_{-1}^{1}\left(\fint_{-1}^{1}\psi(q)\kappa(q-r)\,dq\right)\sqrt{1-r^2}\frac{dr}{r-x} \\
&\quad=\frac{\zeta_1}{\pi^2}\left[\fint_{-1}^{1}\frac{(1-r^2)-(1-x^2)}{\sqrt{1-r^2}}\frac{dr}{r-x}\fint_{-1}^{1}\psi(q)\kappa(q-r)\,dq\right. \\
&\qquad\left.+(1-x^2)\fint_{-1}^{1}\frac{dr}{r-x}\fint_{-1}^{1}\frac{\psi(q)\kappa(q-r)(q-r)}{\sqrt{1-r^2}}\frac{dq}{q-r}\right].
\end{aligned} \tag{4.35}$$

We use the Poincaré-Bertrand formula on the last integral above and note that via l'Hôpital's rule

$$\lim_{\varepsilon\to 0}\kappa(\varepsilon)(\varepsilon)=\lim_{\varepsilon\to 0}\frac{1-\log|\varepsilon|}{\varepsilon^{-2}}=\lim_{\varepsilon\to 0}\frac{-\varepsilon^{-1}}{-2\varepsilon^{-3}}=0. \tag{4.36}$$

This yields

$$\begin{aligned}
&\mathcal{K}\left[\frac{\zeta_1}{\pi}\fint_{-1}^{1}\psi(r)\kappa(r-x)\,dr\right] \\
&\quad=\frac{\zeta_1}{\pi^2}\left[-\fint_{-1}^{1}\frac{r+x}{\sqrt{1-r^2}}\,dr\fint_{-1}^{1}\psi(q)\kappa(q-r)\,dq\right. \\
&\qquad\left.+(1-x^2)\left(-\frac{\pi^2\psi(x)\kappa(x-x)(x-x)}{\sqrt{1-x^2}}+\int_{-1}^{1}dq\fint_{-1}^{1}\frac{\psi(q)\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right)\right] \\
&\quad=\frac{\zeta_1}{\pi^2}\left[-\fint_{-1}^{1}\frac{r+x}{\sqrt{1-r^2}}\,dr\fint_{-1}^{1}\psi(q)\kappa(q-r)\,dq\right. \\
&\qquad\left.+(1-x^2)\fint_{-1}^{1}\psi(q)dq\fint_{-1}^{1}\frac{\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right].
\end{aligned} \tag{4.37}$$

Now note that $\psi(q)[\kappa(q-r)-\kappa(q+r)]$ is odd with respect to q, thus

$$\begin{aligned}\int_{-1}^{1}\psi(q)\kappa(q-r)dq &= \int_{-1}^{1}\psi(q)[\kappa(q-r)-\kappa(q+r)]dq+\int_{-1}^{1}\psi(q)\kappa(q+r)dq\\ &= \int_{-1}^{1}\psi(q)\kappa(q+r)dq, \end{aligned} \tag{4.38}$$

which implies that $\int_{-1}^{1}\psi(q)\kappa(q-r)dq$ is an even function of r. Thus we have

$$\begin{aligned}\mathcal{K}\left[\frac{\zeta_1}{\pi}⨍_{-1}^{1}\psi(r)\kappa(r-x)\,dr\right] = \frac{\zeta_1}{\pi^2}\Bigg[-x⨍_{-1}^{1}\frac{dr}{\sqrt{1-r^2}}⨍_{-1}^{1}\psi(q)\kappa(q-r)\,dq\\ +(1-x^2)⨍_{-1}^{1}\psi(q)dq⨍_{-1}^{1}\frac{\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\Bigg].\end{aligned} \tag{4.39}$$

Thus applying $\mathcal{K}$ to the canonical equation (4.15), using the computations for the individual terms in (4.30) – (4.32), (4.34), and (4.39), yields

$$\begin{aligned}&\gamma_1\zeta_2\psi(x)\sqrt{1-x^2}\\ &\quad= (\sigma+\gamma_0\phi'(1))x-\phi'(1)\frac{\zeta_1}{\pi^2}\left[-x⨍_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\,dr+(1-x^2)⨍_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right]\\ &\qquad-\frac{\gamma_0}{\pi}\left[x⨍_{-1}^{1}\psi(r)\sin^{-1}(r)\,dr+(1-x^2)⨍_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)}⨍_{-1}^{r}\psi(q)\,dq\right]\\ &\qquad+\frac{\zeta_1}{\pi^2}\Bigg[-x⨍_{-1}^{1}\frac{dr}{\sqrt{1-r^2}}⨍_{-1}^{1}\psi(q)\kappa(q-r)\,dq\\ &\qquad\qquad+(1-x^2)⨍_{-1}^{1}\psi(q)dq⨍_{-1}^{1}\frac{\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\Bigg].\end{aligned} \tag{4.40}$$

We combine x and $(1-x^2)$ terms to obtain

$$\begin{aligned}
&\gamma_1\zeta_2\psi(x)\sqrt{1-x^2} \\
&\quad = x\left[\sigma+\gamma_0\phi'(1)+\phi'(1)\frac{\zeta_1}{\pi^2}\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\,dr-\frac{\gamma_0}{\pi}\fint_{-1}^{1}\psi(r)\sin^{-1}(r)\,dr\right. \\
&\qquad \left.-\frac{\zeta_1}{\pi^2}\fint_{-1}^{1}\frac{dr}{\sqrt{1-r^2}}\fint_{-1}^{1}\psi(q)\kappa(q-r)\,dq\right] \\
&\qquad +(1-x^2)\left[-\phi'(1)\frac{\zeta_1}{\pi^2}\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}-\frac{\gamma_0}{\pi}\fint_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)}\fint_{-1}^{r}\psi(q)\,dq\right. \\
&\qquad\qquad \left.+\frac{\zeta_1}{\pi^2}\fint_{-1}^{1}\psi(q)dq\fint_{-1}^{1}\frac{\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right]. \qquad (4.41)
\end{aligned}$$

Finally, appealing to Lemma 4.3 and dividing by $\sqrt{1-x^2}$ yields

$$\begin{aligned}
\gamma_1\zeta_2\psi(x) = \sqrt{1-x^2}\left[-\phi'(1)\frac{\zeta_1}{\pi^2}\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right. \qquad (4.42) \\
\left.-\frac{\gamma_0}{\pi}\fint_{-1}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)}\int_{-1}^{r}\psi(q)\,dq+\frac{\zeta_1}{\pi^2}\int_{-1}^{1}\psi(q)\,dq\fint_{-1}^{1}\frac{\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right],
\end{aligned}$$

which is a Fredholm integral equation of the second kind for $\psi(x)$. □

With these results, we may now proceed with the proof of our main theorem for the existence and uniqueness of solutions that guarantee bounded crack-tip stresses.

Proof of Theorem 4.1. Applying Lemmas 4.2 and 4.4, we see that the original linear integro-differential equation (4.4), subject to conditions (4.6) and (4.7), may be transformed into the Fredholm integral equation (4.42). We can reduce (4.42) to standard form by applying the Poincaré-Bertrand formula to the second integral on the right-hand side. This integral becomes

$$\fint_{-1}^{1}\frac{dr}{r-x}\int_{-1}^{r}\frac{\psi(q)}{\sqrt{1-r^2}}\frac{q-r}{q-r}\,dq=-\pi^2\frac{\psi(x)}{\sqrt{1-x^2}}(x-x)+\int_{-1}^{1}dq\fint_{q}^{1}\frac{\psi(q)}{\sqrt{1-r^2}}\frac{dr}{r-x}. \qquad (4.43)$$

Using this result, we divide (4.42) through by ζ_2 and rearrange terms to obtain

$$\begin{aligned}\gamma_1\psi(x) + \gamma_0\frac{\sqrt{1-x^2}}{\zeta_2\pi}\int_{-1}^{1}\psi(q)\,dq\int_{q}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)} \\ - \zeta_1\frac{\sqrt{1-x^2}}{\zeta_2\pi^2}\int_{-1}^{1}\psi(q)\,dq\fint_{-1}^{1}\frac{\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x} \\ = -\phi'(1)\zeta_1\frac{\sqrt{1-x^2}}{\zeta_2\pi^2}\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}.\end{aligned} \tag{4.44}$$

Thus (4.42) is clearly shown to be a Fredholm integral equation of the second kind in standard form

$$\gamma_1\psi(x) - K\psi(x) = f(x), \tag{4.45}$$

where K is the integral operator

$$K\psi = \int_{-1}^{1}k(x,q)\psi(q)\,dq \tag{4.46}$$

with kernel

$$k(x,q) = \frac{\sqrt{1-x^2}}{\zeta_2\pi}\left[-\gamma_0\int_{q}^{1}\frac{dr}{\sqrt{1-r^2}(r-x)} + \frac{\zeta_1}{\pi}\fint_{-1}^{1}\frac{\kappa(q-r)}{\sqrt{1-r^2}}\frac{dr}{r-x}\right], \tag{4.47}$$

and the right-hand side function is given by

$$f(x) = -\phi'(1)\sqrt{1-x^2}\frac{\zeta_1}{\zeta_2\pi^2}\fint_{-1}^{1}\frac{\lambda(r)}{\sqrt{1-r^2}}\frac{dr}{r-x}. \tag{4.48}$$

Let us assume that K is a Hilbert-Schmidt operator, i.e., $k(x,y) \in L^2([-1,1]^2)$. Then the following hold:

1. K is a compact linear operator and, in particular, has closed range and therefore satisfies the conditions necessary to apply the Fredholm Alternative (see Keener 2000).

2. K has a countable spectrum (see Edmunds and Evans 1987).

Applying the Fredholm Alternative, we see that equation (4.45) has a unique solution $\psi(x)$ in $C[-1,1]$ if and only if γ_1 is not an eigenvalue of K. Since K and hence its spectrum depend on γ_0, this is equivalent to the statement that for each fixed γ_0, a unique, continuous solution to (4.45) exists on $[-1,1]$ for all but countably many values of γ_1. Thus, the proof is concluded after one has shown that K is a Hilbert-Schmidt operator. □

This theorem shows that for each fixed γ_0, in the absence of body forces, the JMB boundary conditions given in (4.3) guarantee bounded crack-tip stresses for all but countably many values of γ_1. In particular, any value of γ_1 outside the spectral radius of K will suffice, where the spectral radius is given by

$$\rho(K) = \max_i\{|\eta_i|\}, \tag{4.49}$$

where $\{\eta_i\}$ are the eigenvalues of K. If K is Hilbert-Schmidt, then K is a bounded linear operator, and thus its spectral radius is bounded by the operator norm of K, which is in turn bounded by the Hilbert-Schmidt norm of K, i.e.,

$$\rho(K) \leq ||K||_{op} \leq ||K||_{HS} = ||k||_{L^2}. \tag{4.50}$$

Thus if we can compute $||k||_{L^2}$, we can generate a threshold value for γ_1. In particular, choosing γ_0 immediately characterizes the values of γ_1 that guarantee bounded crack-

tip stresses, namely

$$|\gamma_1| > ||k(x,q;\gamma_0)||_{L^2([-1,1]^2)} \geq \rho(K;\gamma_0). \tag{4.51}$$

4.2 Square Integrability of the Kernel

Theorem 4.5 *The integral operator K in (4.46) is a Hilbert-Schmidt operator, i.e., $k(x,q)$ is square integrable on $[-1,1]^2$.*

Proof. We use harmonic analysis and complex variable techniques to reduce the kernel to square integrable terms over the domain $S = [-1,1]^2$. The result will be a piecewise definition for the kernel over all of S in which the kernel evaluated on each piece of the domain is either a continuous function or the sum and product of a continuous function and the single logarithmically singular term $\log|x-q|$. Since this term is square integrable on S, this will show that the whole kernel is square integrable.

To begin, we split the kernel (4.47) into three pieces, after appealing to the definition of $\kappa(q-r)$ from (4.9), i.e.,

$$\begin{aligned} k(x,q) &= c_1\sqrt{1-x^2}\int_q^1 \frac{dr}{\sqrt{1-r^2}(r-x)} + c_2\sqrt{1-x^2}\fint_{-1}^1 \frac{(q-r)}{\sqrt{1-r^2}(r-x)}\,dr \\ &\quad - c_2\sqrt{1-x^2}\fint_{-1}^1 \frac{(q-r)\log|q-r|}{\sqrt{1-r^2}(r-x)}\,dr \\ &= T_1(x,q) + T_2(x,q) - T_3(x,q), \end{aligned} \tag{4.52}$$

where $c_1 = c_1(\gamma_0) = -\frac{\gamma_0}{\zeta_2\pi}$ and $c_2 = \frac{\zeta_1}{\zeta_2\pi^2}$. We will consider each term separately in the following subsections. However, we cannot directly evaluate each term over the whole domain S due to the weak singularities that occur in the kernel. Instead, we subdivide the domain S into pieces over which we can isolate a particular weak singularity. The reformulation of the whole kernel will thus be defined piecewise over

these subdomains. Figure 4.1 shows these subdomains, whose total union is the entire domain $S = [-1, 1]^2$, i.e.,

$$S = P \cup L \cup R \cup T \cup B \cup D \cup M \cup I, \tag{4.53}$$

where we have abbreviated numbered subgroups of these divisions by their union, e.g., $T = T_1 \cup T_2$, etc. The explicit definitions of these subdomains are given in Table 4.1. A summary of the value of each term of the kernel over these subdomains will be given at the end of each respective subsection.

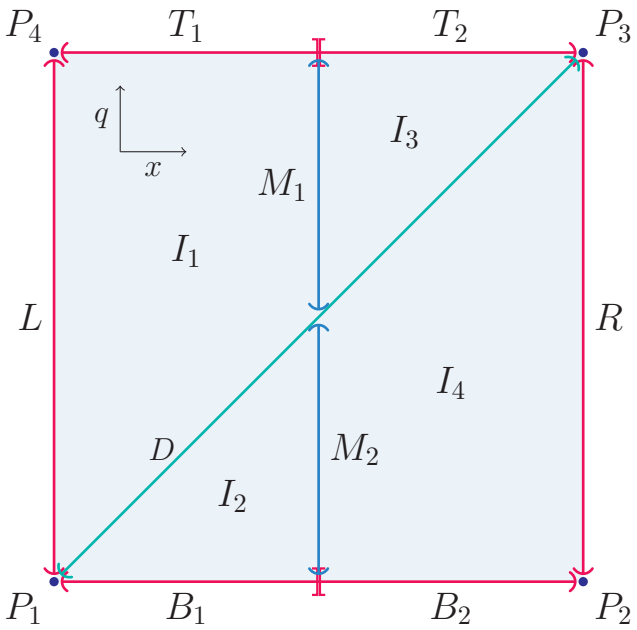


Figure 4.1 Subdomains of the square $S = \{(x, q) \in [-1, 1]^2\}$

Table 4.1 Subdomains of the square $S = [-1, 1]^2$

Subdomain ID	Definition
P_1	$(-1, -1)$
P_2	$(1, -1)$
P_3	$(1, 1)$
P_4	$(-1, 1)$
L	$\{(-1, q) : \lvert q\rvert < 1\}$
R	$\{(1, q) : \lvert q\rvert < 1\}$
T_1	$\{(x, 1) : -1 < x \leq 0\}$
T_2	$\{(x, 1) : 0 \leq x < 1\}$
B_1	$\{(x, -1) : -1 < x \leq 0\}$
B_2	$\{(x, -1) : 0 \leq x < 1\}$
D	$\{(x, x) : \lvert x\rvert < 1\}$
M_1	$\{(0, q) : 0 < q < 1\}$
M_2	$\{(0, q) : -1 < q < 0\}$
I_1	$\{(x, q) : -1 < x < 0,\ \lvert q\rvert < 1,\ x < q\}$
I_2	$\{(x, q) : -1 < x < 0,\ \lvert q\rvert < 1,\ x > q\}$
I_3	$\{(x, q) : 0 < x < 1,\ \lvert q\rvert < 1,\ x < q\}$
I_4	$\{(x, q) : 0 < x < 1,\ \lvert q\rvert < 1,\ x > q\}$

4.2.1 Term 1

Consider the first term of the kernel, given by

$$T_1(x,q) := c_1\sqrt{1-x^2}\,\mathcal{I}_1(x,q), \tag{4.54}$$

where

$$\mathcal{I}_1(x,q) := \fint_q^1 \frac{dr}{\sqrt{1-r^2}(r-x)}, \tag{4.55}$$

is a Cauchy principal value when $q < x < 1$. We will determine the value of $T_1(x,q)$ for any $(x,q) \in S$. The first few cases look at values of x and q on the boundary of S.

Case 1.1: $x \in [-1,1],\ q = 1$ (i.e., $(x,q) \in P_3 \cup P_4 \cup T$)

If $q = 1$, then clearly the integral in (4.55) is zero. Hence, $T_1(x,1) = 0$.

Case 1.2: $x \in (-1,1),\ q = -1$ (i.e., $(x,q) \in B$)

If $q = -1$, then

$$\mathcal{I}_1(x,-1) = \fint_{-1}^1 \frac{dr}{\sqrt{1-r^2}(r-x)} = 0, \tag{4.56}$$

by the Hilbert transform null space result in (4.18). Thus, $T_1(x,-1) = 0$ for all $x \in (-1,1)$.

Case 1.3: $x = -1,\ q \in (-1, 1)$ (i.e., $(x, q) \in L$)

If $x = -1$, then the coefficient $\sqrt{1-x^2}$ will cancel the singularity resulting from the integral, i.e., we have

$$\begin{aligned} T_1(-1, q) &= \lim_{x \to -1} c_1 \sqrt{1-x^2} \fint_q^1 \frac{dr}{\sqrt{1-r^2}(r-x)} \\ &= c_1 \lim_{\varepsilon \to 0} \sqrt{1-(-1+\varepsilon)^2} \int_q^{1-\varepsilon} \frac{dr}{\sqrt{1-r^2}(r+1)} \\ &= c_1 \lim_{\varepsilon \to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left(\frac{-\sqrt{1-r}}{\sqrt{1+r}} \Big|_q^{1-\varepsilon} \right) \\ &= c_1 \lim_{\varepsilon \to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left(\frac{-\sqrt{\varepsilon}}{\sqrt{2-\varepsilon}} + \frac{\sqrt{1-q}}{\sqrt{1+q}} \right) \qquad (4.57) \\ &= c_1 \lim_{\varepsilon \to 0} \left(-\varepsilon + \sqrt{\varepsilon}\sqrt{2-\varepsilon} \frac{\sqrt{1-q}}{\sqrt{1+q}} \right) \\ &= 0. \qquad (4.58) \end{aligned}$$

Case 1.4: $x = q = -1$ (i.e., $(x, q) \in P_1$)

For this case, the method used in Case 1.3, still holds. Substituting $q = -1 + \varepsilon$ into (4.57), we have

$$\begin{aligned} T_1(-1, -1) &= c_1 \lim_{\varepsilon \to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left(\frac{-\sqrt{\varepsilon}}{\sqrt{2-\varepsilon}} + \frac{\sqrt{2-\varepsilon}}{\sqrt{\varepsilon}} \right) \\ &= c_1 \lim_{\varepsilon \to 0} \left(-\varepsilon + (2-\varepsilon) \right) \\ &= 2c_1. \qquad (4.59) \end{aligned}$$

Case 1.5: $x = 1,\ q \in [-1, 1)$ (i.e., $(x, q) \in P_2 \cup R$)

Similarly to the previous case, we have for $q \neq 1$

$$\begin{aligned} T_1(1,q) &= \lim_{x\to 1} c_1\sqrt{1-x^2}\fint_q^1 \frac{dr}{\sqrt{1-r^2}(r-x)} \\ &= c_1 \lim_{\varepsilon\to 0} \sqrt{1-(1-\varepsilon)^2}\int_q^{1-\varepsilon} \frac{dr}{\sqrt{1-r^2}(r-1)} \\ &= -c_1 \lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon}\left(\frac{\sqrt{1+r}}{\sqrt{1-r}}\Big|_q^{1-\varepsilon}\right) \\ &= -c_1 \lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon}\left(\frac{\sqrt{2-\varepsilon}}{\sqrt{\varepsilon}} - \frac{\sqrt{1+q}}{\sqrt{1-q}}\right) \qquad (4.60) \\ &= -c_1 \lim_{\varepsilon\to 0} \left(2-\varepsilon - \sqrt{\varepsilon}\sqrt{2-\varepsilon}\frac{\sqrt{1+q}}{\sqrt{1-q}}\right) \\ &= -2c_1. \qquad (4.61) \end{aligned}$$

Case 1.6: $(x, q) \in (-1, 1)^2,\ x < q$ (i.e., $(x, q) \in M_1 \cup I_1 \cup I_3$)

For this case, there is no singularity at $r = x$. Using integration by parts, we obtain

$$\begin{aligned} \mathcal{I}_1(x,q) &= \fint_q^1 \frac{dr}{\sqrt{1-r^2}(r-x)} \\ &= \frac{\sin^{-1}(r)}{r-x}\Big|_q^1 + \int_q^1 \frac{\sin^{-1}(r)}{(r-x)^2}\,dr \\ &= \frac{\frac{\pi}{2}}{1-x} - \frac{\sin^{-1}(q)}{q-x} + \mathcal{I}_{1\_C6}(x,q), \qquad (4.62) \end{aligned}$$

where the function

$$\mathcal{I}_{1\_C6}(x,q) := \int_q^1 \frac{\sin^{-1}(r)}{(r-x)^2}\,dr \qquad (4.63)$$

has no singularities since $x < q < 1$. Thus $\mathcal{I}_{1\_C6}(x,q)$ is a continuous function of x and q. Substituting this result into (4.54), we have

$$T_1(x,q) = c_1\sqrt{1-x^2}\left[\frac{\pi}{2(1-x)} - \frac{\sin^{-1}(q)}{q-x} + \mathcal{I}_{1\_C6}(x,q)\right], \quad \text{for} \quad -1 < x < q < 1. \tag{4.64}$$

This term is now clearly seen to be a continuous function of x and q in the region $M_1 \cup I_1 \cup I_3$ and hence square integrable.

Case 1.7: $(x,q) \in (-1,1)^2,\ x > q$ (i.e., $(x,q) \in M_2 \cup I_2 \cup I_4$)

For this case, the integrand of $\mathcal{I}_1(x,q)$ has singularities at both $r = 1$ and $r = x$. To compute the integral, we will reformulate it in order to isolate and remove the singularities. The method that we will employ often throughout this section is our standard trick of adding and subtracting a term in the integrand that cancels the singularity. What remains after this cancellation is integrable. We begin by adding and subtracting the term $1/(\sqrt{2}\sqrt{1-r}(r-x))$ to remove the singularity at $r = 1$, i.e.,

$$\mathcal{I}_1(x,q) = \int_q^1 \left(\frac{1}{\sqrt{1+r}} - \frac{1}{\sqrt{2}}\right)\frac{dr}{\sqrt{1-r}(r-x)} + \frac{1}{\sqrt{2}}\int_q^1 \frac{dr}{\sqrt{1-r}(r-x)}. \tag{4.65}$$

Simplifying, we see that the singularity at $r = 1$ is indeed removed from the first integral, since

$$\begin{aligned}
\mathcal{I}_1(x,q) &= \int_q^1 \left(\frac{\sqrt{2}-\sqrt{1+r}}{\sqrt{2}\sqrt{1+r}}\right)\frac{dr}{\sqrt{1-r}(r-x)} + \frac{1}{\sqrt{2}}\int_q^1 \frac{dr}{\sqrt{1-r}(r-x)} \\
&= \int_q^1 \left(\frac{2-(1+r)}{\sqrt{2}\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})}\right)\frac{dr}{\sqrt{1-r}(r-x)} + \frac{1}{\sqrt{2}}\int_q^1 \frac{dr}{\sqrt{1-r}(r-x)} \\
&= \frac{1}{\sqrt{2}}\int_q^1 \frac{\sqrt{1-r}}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})}\frac{dr}{r-x} + \frac{1}{\sqrt{2}}\int_q^1 \frac{dr}{\sqrt{1-r}(r-x)} \\
&= \frac{1}{\sqrt{2}}\int_q^1 \frac{\psi^*(r)\,dr}{r-x} + \frac{1}{\sqrt{2}}\int_q^1 \frac{dr}{\sqrt{1-r}(r-x)},
\end{aligned} \tag{4.66}$$

where

$$\psi^*(r) = \frac{\sqrt{1-r}}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})}. \tag{4.67}$$

We use the same method to remove the singularity at $r = x$ from the first integral in (4.66) by adding and subtracting $\psi^*(x)$ in the numerator. Thus we have

$$\begin{aligned}\mathcal{I}_1(x,q) &= \frac{1}{\sqrt{2}}\int_q^1 \frac{\psi^*(r)-\psi^*(x)}{r-x}\,dr + \frac{\psi^*(x)}{\sqrt{2}}\int_q^1 \frac{dr}{r-x} + \frac{1}{\sqrt{2}}\int_q^1 \frac{dr}{\sqrt{1-r}(r-x)} \\ &= \frac{1}{\sqrt{2}}\Big[f_1(x,q) + \psi^*(x) f_2(x,q) + f_3(x,q)\Big],\end{aligned} \tag{4.68}$$

and we will compute each integral $f_i(x,q)$ individually.

$\underline{f_1(x,q)}$:

For the first integral, we see that the singularity at x is indeed removed since

$$f_1(x,q) := \int_q^1 \frac{\psi^*(r)-\psi^*(x)}{r-x}\,dr \tag{4.69}$$

$$\begin{aligned}&= \int_q^1 \left[\frac{\sqrt{1-r}}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})} - \frac{\sqrt{1-x}}{\sqrt{1+x}(\sqrt{2}+\sqrt{1+x})}\right]\frac{dr}{r-x} \\ &= \int_q^1 \left[\frac{\sqrt{1-r}\sqrt{1+x}(\sqrt{2}+\sqrt{1+x}) - \sqrt{1-x}\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})\sqrt{1+x}(\sqrt{2}+\sqrt{1+x})}\right]\frac{dr}{r-x} \\ &= \frac{\sqrt{2}}{\sqrt{1+x}(\sqrt{2}+\sqrt{1+x})}\int_q^1 \frac{\sqrt{1-r}\sqrt{1+x} - \sqrt{1-x}\sqrt{1+r}}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})}\frac{dr}{r-x} \\ &\quad + \frac{1}{\sqrt{1+x}(\sqrt{2}+\sqrt{1+x})}\int_q^1 \frac{\sqrt{1-r}(1+x) - \sqrt{1-x}(1+r)}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})}\frac{dr}{r-x},\end{aligned} \tag{4.70}$$

and we set $C(x) = \sqrt{1+x}(\sqrt{2}+\sqrt{1+x})$ to obtain

$$\begin{aligned} f_1(x,q) &= \frac{\sqrt{2}}{C(x)}\fint_q^1 \frac{(1-r)(1+x)-(1-x)(1+r)}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})(\sqrt{1-r}\sqrt{1+x}+\sqrt{1-x}\sqrt{1+r})}\frac{dr}{r-x} \\ &\quad + \frac{1}{C(x)}\fint_q^1 \frac{(1-r)(1+x)^2-(1-x)(1+r)^2}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})(\sqrt{1-r}(1+x)+\sqrt{1-x}(1+r))}\frac{dr}{r-x} \\ &= \frac{-2\sqrt{2}}{C(x)}\int_q^1 \frac{dr}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})(\sqrt{1-r}\sqrt{1+x}+\sqrt{1-x}\sqrt{1+r})} \\ &\quad + \frac{1}{C(x)}\int_q^1 \frac{[r(x-1)-(x+3)]\,dr}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})(\sqrt{1-r}(1+x)+\sqrt{1-x}(1+r))}. \end{aligned} \tag{4.71}$$

Each of the integrands on the right-hand side is continuous everywhere on $[q,1]$ since $r \geq q > -1$ and $x \in (q,1)$. Thus their integrals exist and are continuous on $M_2 \cup I_2 \cup I_4$. Denoting these integrals by $f_{1a}(x,q)$ and $f_{1b}(x,q)$, respectively, we have

$$f_1(x,q) = \frac{1}{\sqrt{1+x}(\sqrt{2}+\sqrt{1+x})}\left[-2\sqrt{2}\,f_{1a}(x,q) + f_{1b}(x,q)\right]. \tag{4.72}$$

$\underline{f_2(x,q)}$:

From the definition of a Cauchy principal value, we have

$$\begin{aligned} f_2(x,q) &:= \fint_q^1 \frac{dr}{r-x} \qquad (4.73) \\ &:= \lim_{\varepsilon\to 0}\left[\int_q^{x-\varepsilon}\frac{dr}{r-x} + \int_{x+\varepsilon}^1 \frac{dr}{r-x}\right] \\ &= \lim_{\varepsilon\to 0}\left[\log|r-x|\Big|_q^{x-\varepsilon} + \log|r-x|\Big|_{x+\varepsilon}^1\right] \\ &= \lim_{\varepsilon\to 0}\left[\log|\varepsilon| - \log|q-x| + \log|1-x| - \log|\varepsilon|\right] \\ &= \log|1-x| - \log|q-x| \\ &= \log(1-x) - \log(x-q), \qquad (4.74) \end{aligned}$$

which is a continuous function of x and q since $x \in (q, 1)$.

$\underline{f_3(x,q)}$:

The third integral, defined by

$$f_3(x,q) := \int_q^1 \frac{dr}{\sqrt{1-r}(r-x)}, \tag{4.75}$$

may be rewritten as $f_3(x,q) = 2\pi\Phi(x,q)$, where

$$\Phi(x,q) := \frac{1}{2\pi i}\int_q^1 \frac{dr}{(r-1)^{1/2}(r-x)}. \tag{4.76}$$

Muskhelishvili (1977, §29 – §30) shows that in a neighborhood of the point $x = 1$, the function $\Phi(x,q)$, for $x \in (q,1)$, behaves like

$$\Phi(x,q) = -\frac{e^{-\frac{\pi}{2}i}}{2i\sin\frac{\pi}{2}}(x-1)^{-\frac{1}{2}} + B_0 + B_1(x-1) + B_2(x-1)^2 + \cdots . \tag{4.77}$$

Thus the singularity in $\Phi(x,q)$ would be removed if it was multiplied by $(x-1)^{1/2}$, or equivalently by $(1-x)^{1/2}$. Therefore, we compute

$$f_3^*(x,q) = \sqrt{1-x}\, f_3(x,q). \tag{4.78}$$

Again by adding and subtracting terms, we first cancel the singularity at $r = x$, i.e.,

$$\begin{aligned} f_3^*(x,q) &:= \fint_q^1 \frac{\sqrt{1-x}}{\sqrt{1-r}(r-x)}\,dr \\ &= \fint_q^1 \frac{\sqrt{1-x}-\sqrt{1-r}}{\sqrt{1-r}(r-x)}\,dr + \int_q^1 \frac{\sqrt{1-r}}{\sqrt{1-r}(r-x)}\,dr \\ &= \fint_q^1 \frac{(1-x)-(1-r)}{\sqrt{1-r}(\sqrt{1-r}+\sqrt{1-x})}\frac{dr}{r-x} + \int_q^1 \frac{dr}{r-x} \\ &= \fint_q^1 \frac{dr}{\sqrt{1-r}(\sqrt{1-r}+\sqrt{1-x})} + \int_q^1 \frac{dr}{r-x} \\ &= f_4(x,q) + f_2(x,q), \end{aligned} \tag{4.79}$$

where we note that the second integral on the right-hand side is exactly $f_2(x,q)$, which we calculated above in (4.74). For the first integral, which we denote $f_4(x,q)$, we add and subtract $1/(\sqrt{1-x}\sqrt{1-r})$ to remove the singularity at $r = 1$. This yields

$$f_4(x,q) := \fint_q^1 \frac{dr}{\sqrt{1-r}\left(\sqrt{1-r}+\sqrt{1-x}\right)} \tag{4.80}$$

$$\begin{aligned} &= \fint_q^1 \left[\frac{1}{\left(\sqrt{1-r}+\sqrt{1-x}\right)} - \frac{1}{\sqrt{1-x}}\right]\frac{dr}{\sqrt{1-r}} + \frac{1}{\sqrt{1-x}}\int_q^1 \frac{dr}{\sqrt{1-r}} \\ &= \fint_q^1 \left[\frac{\sqrt{1-x}-\left(\sqrt{1-r}+\sqrt{1-x}\right)}{\sqrt{1-x}\left(\sqrt{1-r}+\sqrt{1-x}\right)}\right]\frac{dr}{\sqrt{1-r}} + \frac{1}{\sqrt{1-x}}\left(-2\sqrt{1-r}\,\Big|_q^1\right) \\ &= -\frac{1}{\sqrt{1-x}}\int_q^1 \frac{dr}{\sqrt{1-r}+\sqrt{1-x}} + \frac{1}{\sqrt{1-x}}\left(2\sqrt{1-q}\right). \end{aligned} \tag{4.81}$$

The remaining integral has an exact antiderivative, given by

$$\begin{aligned}\int_q^1 \frac{dr}{\sqrt{1-r}+\sqrt{1-x}} &= \Big[-2\sqrt{1-r}+2\sqrt{1-x}\log\big(\sqrt{1-r}+\sqrt{1-x}\big)\Big]\Big|_q^1 \\ &= 2\sqrt{1-x}\log(\sqrt{1-x}) - \Big(-2\sqrt{1-q}+2\sqrt{1-x}\log(\sqrt{1-q}+\sqrt{1-x})\Big) \\ &= \sqrt{1-x}\log(1-x)+2\sqrt{1-q}-2\sqrt{1-x}\log(\sqrt{1-q}+\sqrt{1-x}). \qquad (4.82)\end{aligned}$$

Substituting this into (4.81) yields

$$\begin{aligned}f_4(x,q) &= -\log(1-x) - 2\frac{\sqrt{1-q}}{\sqrt{1-x}} + 2\log(\sqrt{1-q}+\sqrt{1-x}) + 2\frac{\sqrt{1-q}}{\sqrt{1-x}} \\ &= 2\log(\sqrt{1-q}+\sqrt{1-x}) - \log(1-x). \qquad (4.83)\end{aligned}$$

We apply this result, combined with (4.74), to (4.79) to obtain

$$f_3^*(x,q) = 2\log(\sqrt{1-q}+\sqrt{1-x}) - \log(x-q). \qquad (4.84)$$

Recalling how we split up the integral $\mathcal{I}_1$ in (4.68), we substitute in the values of f_1, f_2, and ψ^* from (4.72), (4.74), and (4.67), respectively, to obtain

$$\begin{aligned}\mathcal{I}_1(x,q) &= \frac{1}{\sqrt{2}}\left[f_1(x,q)+\psi^*(x)f_2(x,q)+f_3(x,q)\right] \\ &= \frac{1}{\sqrt{2}}\left[\frac{1}{\sqrt{1+x}(\sqrt{2}+\sqrt{1+x})}\Big(-2\sqrt{2}f_{1a}(x,q)+f_{1b}(x,q)\Big)\right. \\ &\qquad \left. + \frac{\sqrt{1-x}}{\sqrt{1+x}(\sqrt{2}+\sqrt{1+x})}\Big(\log(1-x)-\log(x-q)\Big)+f_3(x,q)\right]. \qquad (4.85)\end{aligned}$$

Finally, we appeal to the definition of Term 1 in (4.54), combining this result with (4.78) and (4.84). This yields, for $(x,q) \in (-1,1)^2,\ x > q$,

$$\begin{aligned} T_1(x,q) = \frac{c_1}{\sqrt{2}} \Bigg[& \frac{\sqrt{1-x}}{(\sqrt{2}+\sqrt{1+x})} \Big(-2\sqrt{2} f_{1a}(x,q) + f_{1b}(x,q) \Big) \\ & + \frac{(1-x)}{(\sqrt{2}+\sqrt{1+x})} \Big(\log(1-x) - \log(x-q) \Big) \\ & + \sqrt{1+x} \Big(2\log(\sqrt{1-q}+\sqrt{1-x}) - \log(x-q) \Big) \Bigg], \end{aligned} \tag{4.86}$$

which is clearly continuous, since we already showed f_{1a} and f_{1b} are both continuous functions.

Case 1.8: $x = q \neq \pm 1$ (i.e., $(x,q) \in D$)

For this case, we try to isolate the singularity at $r = x = q$. First, we split the integral into two pieces:

$$\mathcal{I}_1(x,q) = \int_q^{\frac{1+q}{2}} \frac{dr}{\sqrt{1-r^2}(r-x)} + \int_{\frac{1+q}{2}}^1 \frac{dr}{\sqrt{1-r^2}(r-x)}. \tag{4.87}$$

There is no singularity at $r = x = q$ in the second piece and we use integration by parts to obtain

$$\begin{aligned} \int_{\frac{1+q}{2}}^1 \frac{dr}{\sqrt{1-r^2}(r-x)} &= \frac{\sin^{-1}(r)}{r-x} \Big|_{\frac{1+q}{2}}^1 + \int_{\frac{1+q}{2}}^1 \frac{\sin^{-1}(r)}{(r-x)^2}\, dr \\ &= \frac{\frac{\pi}{2}}{1-x} - \frac{\sin^{-1}(\frac{1+q}{2})}{\frac{1+q}{2}-x} + \mathcal{I}_{1\_C8a}(x,q) \\ &= \frac{\pi}{2(1-x)} - \frac{2\sin^{-1}(\frac{1+q}{2})}{1+q-2x} + \mathcal{I}_{1\_C8a}(x,q), \end{aligned} \tag{4.88}$$

where the integral $\mathcal{I}_{1\_C8a}(x,q)$ is a continuous function of $x = q < \frac{1+q}{2}$.

In the first piece, there is no longer a singularity at $r = 1$ and we apply our standard trick to try to remove the singularity at $r = x$. This yields

$$\begin{aligned}
&\int_q^{\frac{1+q}{2}} \frac{dr}{\sqrt{1-r^2}(r-x)} \\
&= \int_q^{\frac{1+q}{2}} \left[\frac{1}{\sqrt{1-r^2}} - \frac{1}{\sqrt{1-x^2}}\right] \frac{dr}{r-x} + \frac{1}{\sqrt{1-x^2}} \int_q^{\frac{1+q}{2}} \frac{dr}{r-x} \\
&= \frac{1}{\sqrt{1-x^2}} \int_q^{\frac{1+q}{2}} \frac{\sqrt{1-x^2}-\sqrt{1-r^2}}{\sqrt{1-r^2}} \frac{dr}{r-x} + \frac{1}{\sqrt{1-x^2}} \log|r-x| \Big|_q^{\frac{1+q}{2}} \\
&= \frac{1}{\sqrt{1-x^2}} \int_q^{\frac{1+q}{2}} \frac{r+x}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\, dr \\
&\quad + \frac{1}{\sqrt{1-x^2}} \Big(\log\left|\tfrac{1+q}{2} - x\right| - \log|q-x|\Big) \\
&= \frac{1}{\sqrt{1-x^2}} \mathcal{I}_{1_C8b}(x,q) + \frac{1}{\sqrt{1-x^2}} \Big(\log\left|\tfrac{1+q}{2} - x\right| - \log|q-x|\Big),
\end{aligned} \tag{4.89}$$

where the integral $\mathcal{I}_{1\_C8b}(x,q)$ is a continuous function of $x = q \neq \pm 1$. However, we are left with a logarithmic singularity at $x = q$ in the last term. Fortunately, the logarithm function is still square integrable, as is its sum and product with a continuous function. Applying these results to (4.54), the full term becomes

$$\begin{aligned}
T_1(x,q) = c_1 &\left[\mathcal{I}_{1_C8b}(x,q) + \log\left|\tfrac{1+q}{2} - x\right| - \log|q-x|\right] \\
&+ c_1\sqrt{1-x^2}\left[\frac{\pi}{2(1-x)} - \frac{2\sin^{-1}(\frac{1+q}{2})}{1+q-2x} + \mathcal{I}_{1_C8a}(x,q)\right],
\end{aligned} \tag{4.90}$$

which we have shown to be in $L^2(D)$.

Thus we have computed Term 1 for all the subdomains of $S = [-1,1]^2$ as shown in Figure 4.1. A summary of these cases is given below in Table 4.2.

Table 4.2 Summary of the piecewise values of Term 1 of the kernel on $S = [-1, 1]^2$

Domain	$T_1(x, q)$
P_1	$2c_1$
$P_2 \cup R$	$-2c_1$
$P_3 \cup P_4 \cup L \cup T \cup B$	0
D	$c_1\Big[\mathcal{I}_{1\_C8b}(x,q) + \log\left\lvert\frac{1+q}{2} - x\right\rvert - \log\lvert q - x\rvert\Big]$ $+c_1\sqrt{1-x^2}\Big[\frac{\pi}{2(1-x)} - \frac{2\sin^{-1}(\frac{1+q}{2})}{1+q-2x} + \mathcal{I}_{1\_C8a}(x,q)\Big]$
$M_1 \cup I_1 \cup I_3$	$c_1\sqrt{1-x^2}\left[\frac{\pi}{2(1-x)} - \frac{\sin^{-1}(q)}{q-x} + \mathcal{I}_{1\_C6}(x,q)\right]$
$M_2 \cup I_2 \cup I_4$	$\frac{c_1}{\sqrt{2}}\Big[\frac{\sqrt{1-x}}{(\sqrt{2}+\sqrt{1+x})}\left(-2\sqrt{2}f_{1a}(x,q) + f_{1b}(x,q)\right)$ $+\frac{(1-x)}{(\sqrt{2}+\sqrt{1+x})}\Big(\log(1-x) - \log(x-q)\Big)$ $+\sqrt{1+x}\left(2\log(\sqrt{1-q}+\sqrt{1-x}) - \log(x-q)\right)\Big]$

4.2.2 Term 2

We reformulate the second term of the kernel in the same manner as the first. Recall from (4.52) that this term is given by

$$T_2(x, q) = c_2\sqrt{1-x^2}\,\mathcal{I}_2(x, q), \tag{4.91}$$

where

$$\mathcal{I}_2(x, q) = \fint_{-1}^{1} \frac{q-r}{\sqrt{1-r^2}(r-x)}\,dr. \tag{4.92}$$

As above, we first consider values of x and q on the boundary of S.

Case 2.1: $x = -1,\ q \in (-1, 1]$ (i.e., $(x, q) \in P_4 \cup L$)

For this case, the singularity at $r = x = -1$ that occurs in the integral will be canceled by the coefficient $\sqrt{1-x^2}$, similarly to the computation of $f_3(x, q)$ above.

$$\begin{aligned}
T_2(-1, q) &= \lim_{x\to -1} c_2\sqrt{1-x^2} \fint_{-1}^{1} \frac{q-r}{\sqrt{1-r^2}(r-x)}\, dr \\
&= c_2 \lim_{\varepsilon\to 0} \sqrt{1-(-1+\varepsilon)^2} \int_{-1+\varepsilon}^{1-\varepsilon} \frac{q-r}{\sqrt{1-r^2}(r+1)}\, dr \\
&= c_2 \lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left[\int_{-1+\varepsilon}^{1-\varepsilon} \frac{(q-r)-(q+1)}{\sqrt{1-r^2}(1+r)}\, dr + (q+1) \int_{-1+\varepsilon}^{1-\varepsilon} \frac{dr}{\sqrt{1-r^2}(1+r)} \right] \\
&= c_2 \lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left[-\int_{-1}^{1} \frac{dr}{\sqrt{1-r^2}} + (1+q) \left(\frac{-\sqrt{1-r}}{\sqrt{1+r}} \Big|_{-1+\varepsilon}^{1-\varepsilon} \right) \right] \\
&= c_2 \lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left[-\pi + (1+q) \left(-\frac{\sqrt{\varepsilon}}{\sqrt{2-\varepsilon}} + \frac{\sqrt{2-\varepsilon}}{\sqrt{\varepsilon}} \right) \right] \\
&= 2c_2(1+q). \qquad (4.93)
\end{aligned}$$

Case 2.2: $x = 1,\ q \in [-1, 1)$ (i.e., $(x, q) \in P_2 \cup R$)

Similarly to the previous case, we have

$$\begin{aligned}
T_2(1, q) &= c_2 \lim_{\varepsilon\to 0} \sqrt{1-(1-\varepsilon)^2} \int_{-1+\varepsilon}^{1-\varepsilon} \frac{q-r}{\sqrt{1-r^2}(r-1)}\, dr \\
&= -c_2 \lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left[\int_{-1+\varepsilon}^{1-\varepsilon} \frac{(q-r)-(q-1)}{\sqrt{1-r^2}(1-r)}\, dr + (q-1) \int_{-1+\varepsilon}^{1-\varepsilon} \frac{dr}{\sqrt{1-r^2}(1-r)} \right] \\
&= -c_2 \lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left[\int_{-1}^{1} \frac{dr}{\sqrt{1-r^2}} - (1-q) \left(\frac{\sqrt{1+r}}{\sqrt{1-r}} \Big|_{-1+\varepsilon}^{1-\varepsilon} \right) \right] \\
&= -c_2 \lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left[\pi - (1-q) \left(\frac{\sqrt{2-\varepsilon}}{\sqrt{\varepsilon}} - \frac{\sqrt{\varepsilon}}{\sqrt{2-\varepsilon}} \right) \right] \\
&= 2c_2(1-q). \qquad (4.94)
\end{aligned}$$

Case 2.3: $x \in (-1,1),\ q \in [-1,1],\ x \neq q$ (i.e., $(x,q) \in T \cup B \cup M \cup I$)

We apply our standard trick to remove the singularity at $r = x$ to obtain

$$\begin{aligned}\mathcal{I}_2(x,q) &= \fint_{-1}^{1} \frac{(q-r)-(q-x)}{\sqrt{1-r^2}} \frac{dr}{r-x} + (q-x) \fint_{-1}^{1} \frac{dr}{\sqrt{1-r^2}(r-x)} \\ &= -\int_{-1}^{1} \frac{dr}{\sqrt{1-r^2}} \\ &= -\pi, \end{aligned} \tag{4.95}$$

where we have used the result in (4.18) to eliminate the last term. Appealing to (4.91) yields

$$T_2(x,q) = -c_2 \pi \sqrt{1-x^2}. \tag{4.96}$$

Case 2.4: $x = q$ (i.e., $(x,q) \in P_1 \cup P_3 \cup D$)

For $x = q$, we have

$$\begin{aligned}\mathcal{I}_2(x,x) &= \fint_{-1}^{1} \frac{x-r}{\sqrt{1-r^2}(r-x)}\, dr \\ &= -\int_{-1}^{1} \frac{dr}{\sqrt{1-r^2}} \\ &= -\sin^{-1}(r)\,\Big|_{-1}^{1} \\ &= -\pi. \end{aligned} \tag{4.97}$$

Note that this holds even for $x = q = \pm 1$. Thus by (4.91), we have

$$\begin{aligned} T_2(x,x) &= -c_2 \pi \sqrt{1-x^2} \\ &= \begin{cases} 0, & \text{for } \ x = \pm 1 \\ -c_2 \pi \sqrt{1-x^2}, & \text{for } \ x \in (-1,1). \end{cases} \end{aligned} \tag{4.98}$$

Table 4.3 Summary of the piecewise values of Term 2 of the kernel on $S = [-1, 1]^2$

Subdivision	$T_2(x, q)$
$P_1 \cup P_3$	0
$P_2 \cup R$	$2c_2(1 - q)$
$P_4 \cup L$	$2c_2(1 + q)$
$T \cup B \cup D \cup M \cup I$	$-c_2\pi\sqrt{1 - x^2}$

These cases are summarized in Table 4.3. It is clear that each of these functions is continuous over its domain of definition.

4.2.3 Term 3

Recall from (4.52) that the third term of the kernel is given by

$$T_3(x, q) = c_2\sqrt{1 - x^2}\, \mathcal{I}_3(x, q), \tag{4.99}$$

where

$$\mathcal{I}_3(x, q) = \fint_{-1}^{1} \frac{(q - r)\log|q - r|}{\sqrt{1 - r^2}(r - x)}\, dr. \tag{4.100}$$

When $x \neq q$, we can use our standard trick to remove the singularity at $r = x$, which yields

$$\begin{aligned}
\mathcal{I}_3(x, q) &= \fint_{-1}^{1} \frac{[(q - r) - (q - x)]\log|q - r|}{\sqrt{1 - r^2}(r - x)}\, dr + (q - x)\fint_{-1}^{1} \frac{\log|q - r|}{\sqrt{1 - r^2}(r - x)}\, dr \\
&= -\fint_{-1}^{1} \frac{\log|q - r|}{\sqrt{1 - r^2}}\, dr + (q - x)\fint_{-1}^{1} \frac{\log|q - r|}{\sqrt{1 - r^2}(r - x)}\, dr \\
&= -g_1(q) + (q - x)g_2(x, q),
\end{aligned} \tag{4.101}$$

where

$$g_1(q) := \fint_{-1}^{1} \frac{\log|q-r|}{\sqrt{1-r^2}}\, dr, \tag{4.102}$$

and

$$g_2(x,q) := \fint_{-1}^{1} \frac{\log|q-r|}{\sqrt{1-r^2}(r-x)}\, dr. \tag{4.103}$$

For $x = q$ (i.e., $(x,q) \in P_1 \cup P_3 \cup D$), we simply have

$$\begin{aligned} \mathcal{I}_3(q,q) &= \fint_{-1}^{1} \frac{(q-r)\log|q-r|}{\sqrt{1-r^2}(r-q)}\, dr \\ &= -\fint_{-1}^{1} \frac{\log|q-r|}{\sqrt{1-r^2}}\, dr = -g_1(q). \end{aligned} \tag{4.104}$$

Thus for any $x, q \in [-1,1]$ we have

$$\mathcal{I}_3(x,q) = \begin{cases} -g_1(q), & x = q, \\ -g_1(q) + (q-x)g_2(x,q), & x \neq q, \end{cases} \tag{4.105}$$

and therefore

$$T_3(x,q) = \begin{cases} -c_2\sqrt{1-x^2}\, g_1(q), & x = q, \\ -c_2\sqrt{1-x^2}\, g_1(q) + c_2(q-x)\sqrt{1-x^2}\, g_2(x,q), & x \neq q. \end{cases} \tag{4.106}$$

So to compute $\mathcal{I}_3(x,q)$, it is sufficient to compute the values of $g_1(q)$ and $g_2(x,q)$. Note from (4.105) that we need only compute $g_2(x,q)$ on $S\backslash(P_1 \cup P_3 \cup D)$. By defining this term to be identically zero on $P_1 \cup P_3 \cup D$, we may combine these cases so that the third term of the kernel becomes simply

$$T_3(x,q) = -c_2\sqrt{1-x^2}\, g_1(q) + c_2(q-x)\sqrt{1-x^2}\, g_2(x,q). \tag{4.107}$$

Computing $g_1(q)$

Note that if $q \in (-1, 1)$, then the derivative of $g_1(q)$ (4.102) vanishes, i.e.,

$$\frac{\partial g_1(q)}{\partial q} = -\fint_{-1}^{1} \frac{dr}{\sqrt{1-r^2}(r-q)} = 0, \tag{4.108}$$

by the null space result in (4.18). Thus g_1 is independent of q, so we may compute its value at any particular $q \in (-1, 1)$, say $q = 0$. For this value, integration by parts yields

$$\begin{aligned} g_1(0) &= \fint_{-1}^{1} \frac{\log|r|}{\sqrt{1-r^2}}\, dr \\ &= \sin^{-1}(r) \log|r| \Big|_{-1}^{1} - \int_{-1}^{1} \frac{\sin^{-1}(r)}{r}\, dr \\ &= -\int_{-1}^{1} \frac{\sin^{-1}(r)}{r}\, dr. \end{aligned} \tag{4.109}$$

Note that the integrand is continuous over $[-1, 1]$, since via l'Hôpital's rule

$$\lim_{r\to 0} \frac{\sin^{-1}(r)}{r} = \lim_{r\to 0} \frac{(1-r)^{-1/2}}{1} = 1. \tag{4.110}$$

Thus the value of this integral is a finite constant, which is clearly in $L^2([-1, 1]^2)$. In particular, we use the `int` function in MATLAB® (The MathWorks, Inc. 2012) to evaluate this integral, which yields

$$\int_{-1}^{1} \frac{\sin^{-1}(r)}{r}\, dr = \pi \log(2). \tag{4.111}$$

If $q = \pm 1$, we cannot use the null space result. However, we can still show that the integral is continuous. In particular, we will isolate the singularities at $r = \pm 1$ and

use the fact that

$$\lim_{r\to p}\sqrt{p-r}\log|p-r| = \lim_{r\to p}\frac{-\frac{1}{p-r}}{\frac{1}{2}(p-r)^{-3/2}} = \lim_{r\to p} -2\sqrt{p-r} = 0, \tag{4.112}$$

which holds for any $p \in [-1,1]$. Thus for $q = 1$, we have

$$\begin{aligned}
g_1(1) &= \fint_{-1}^{1} \frac{\log|1-r|}{\sqrt{1-r^2}}\,dr \\
&= \fint_{-1}^{0} \frac{\log|1-r|}{\sqrt{1-r^2}}\,dr + \fint_{0}^{1} \frac{\log|1-r|}{\sqrt{1-r}\sqrt{1+r}}\,dr \\
&= \sin^{-1}(r)\log|1-r|\,\Big|_{-1}^{0} + \int_{-1}^{0}\frac{\sin^{-1}(r)}{1-r}\,dr - 2\sqrt{1-r}\,\frac{\log|1-r|}{\sqrt{1+r}}\,\Big|_{0}^{1} \\
&\qquad + 2\int_{0}^{1}\sqrt{1-r}\left(\frac{-1}{(1-r)\sqrt{1+r}} - \frac{\log|1-r|}{2(1+r)^{3/2}}\right)dr \\
&= \frac{\pi}{2}\log 2 + \int_{-1}^{0}\frac{\sin^{-1}(r)}{1-r}\,dr - 0 - 2\int_{0}^{1}\frac{dr}{\sqrt{1-r^2}} - \int_{0}^{1}\frac{\sqrt{1-r}\log|1-r|}{(1+r)^{3/2}}\,dr \\
&= \frac{\pi}{2}(\log 2 - 2) + \int_{-1}^{0}\frac{\sin^{-1}(r)}{1-r}\,dr - \int_{0}^{1}\frac{\sqrt{1-r}\log|1-r|}{(1+r)^{3/2}}\,dr,
\end{aligned} \tag{4.113}$$

where both the remaining integrands are continuous and hence integrate to finite constants. Thus $g_1(1)$ is a finite constant and clearly square integrable. A similar computation shows that $g_1(-1)$ is also in L^2. Again, we use MATLAB®'s `int` function to compute the actual value, given by

$$g_1(\pm 1) = \fint_{-1}^{1}\frac{\log|\pm 1 - r|}{\sqrt{1-r^2}}\,dr = -\pi\log(2). \tag{4.114}$$

Thus for all $q \in [-1,1]$, we have

$$g_1(q) = -\pi\log(2). \tag{4.115}$$

By our reformulation of Term 3 in (4.107), this already yields all the values of this term when $x = q$, i.e.,

$$T_3(x,q) = c_2\pi \log(2)\sqrt{1-x^2}, \ \ \forall \ \ (x,q) \in P_1 \cup P_3 \cup D. \tag{4.116}$$

Computing $g_2(x,q)$

Recall from (4.103) that g_2 is given by

$$g_2(x,q) := \fint_{-1}^{1} \frac{\log|q-r|}{\sqrt{1-r^2}(r-x)}\, dr. \tag{4.117}$$

Before considering cases for specific values of x and q, we make a few observations.

First, we note that g_2 satisfies the symmetry property

$$g_2(-x,-q) = -g_2(x,q), \tag{4.118}$$

since by a change of variables we have

$$\begin{aligned} g_2(x,q) &= \fint_{1}^{-1} \frac{\log|q+s|}{\sqrt{1-s^2}(-s-x)}(-ds) \\ &= -\fint_{-1}^{1} \frac{\log|-q-s|}{\sqrt{1-s^2}(s-(-x))} ds \\ &= -g_2(-x,-q). \end{aligned} \tag{4.119}$$

Second, we note that $g_2(x,q)$ is independent of q for $(x,q) \in (-1,1)^2$, $x \neq q$, since

$$\begin{aligned}\frac{\partial g_2(x,q)}{\partial q} &= \fint_{-1}^{1} \frac{dr}{\sqrt{1-r^2}(r-x)(q-r)} \\ &= \fint_{-1}^{1} \frac{1}{\sqrt{1-r^2}} \left[\frac{1}{r-x} + \frac{1}{q-r}\right] \frac{1}{q-x}\, dr \\ &= \frac{1}{q-x}\left[\fint_{-1}^{1} \frac{dr}{\sqrt{1-r^2}(r-x)} - \fint_{-1}^{1} \frac{dr}{\sqrt{1-r^2}(r-q)}\right] \\ &= 0, \end{aligned} \tag{4.120}$$

since for these values of x and q, we may apply the null space result (4.18) as before. Thus $g_2(x,q)$ is independent of q, so we may replace q by any value in $(-1,1)$, say $q = 0$ (except when $x = 0$; this will be a separate case below). This gives us

$$g_2(x,q) = g_2(x,0) = \fint_{-1}^{1} \frac{\log|r|}{\sqrt{1-r^2}(r-x)}\, dr, \quad x, q \in (-1,1),\ x \notin \{0,q\}. \tag{4.121}$$

This integral is computed in the first of the value-specific cases for $(x,q) \in S$ below.

Case 3.1: $x \in (0,1),\ q = 0$

By the observation above, we see that this case provides values for all $x \in (0,1)$ and $q \in (-1,1)$ (for $x \neq q$), i.e., $(x,q) \in I_3 \cup I_4$. By the symmetry property (4.118), this additionally covers values of g_2 for all $x \in (-1,0)$ and $q \in (-1,1)$ (for $x \neq q$), i.e., $(x,q) \in I_1 \cup I_2$. To compute $g_2(x,0)$, we first try to isolate each singularity. Breaking up the interval of integration, we have

$$\begin{aligned} g_2(x,0) &:= \fint_{-1}^{1} \frac{\log|r|}{\sqrt{1-r^2}(r-x)}\, dr \\ &= \fint_{-1}^{\frac{x}{2}} \frac{\log|r|}{\sqrt{1-r^2}(r-x)}\, dr + \int_{\frac{x}{2}}^{\frac{1+x}{2}} \frac{\log|r|}{\sqrt{1-r^2}(r-x)}\, dr + \int_{\frac{1+x}{2}}^{1} \frac{\log|r|}{\sqrt{1-r^2}(r-x)}\, dr. \end{aligned} \tag{4.122}$$

We will consider each integral separately. For the first, we integrate by parts to remove the singularity at $r = -1$, i.e.,

$$\begin{aligned}
\fint_{-1}^{\frac{x}{2}} \frac{\log|r|}{\sqrt{1-r^2}(r-x)}\,dr &= \frac{\sin^{-1}(r)\log|r|}{r-x}\Big|_{-1}^{\frac{x}{2}} - \int_{-1}^{\frac{x}{2}} \sin^{-1}(r)\left[\frac{1}{r(r-x)} - \frac{\log|r|}{(r-x)^2}\right] dr \\
&= \frac{\sin^{-1}(\frac{x}{2})\log\left|\frac{x}{2}\right|}{\frac{x}{2}-x} - \int_{-1}^{\frac{x}{2}} \frac{\sin^{-1}(r)}{r(r-x)}\,dr + \int_{-1}^{\frac{x}{2}} \frac{\sin^{-1}(r)\log|r|}{(r-x)^2}\,dr \\
&= -\frac{2}{x}\sin^{-1}\left(\frac{x}{2}\right)\log\left|\frac{x}{2}\right| - g_{2_C1a}(x) + g_{2_C1b}(x). \qquad (4.123)
\end{aligned}$$

The last two integrals, which we denote $g_{2\_C1a}(x)$ and $g_{2\_C1b}(x)$, are continuous by (4.110) and by (again using l'Hôpital's rule)

$$\lim_{r\to 0}\frac{\log|r|}{(\sin^{-1}(r))^{-1}} = \lim_{r\to 0}\frac{\frac{1}{r}}{-(\sin^{-1}(r))^{-2}\frac{1}{\sqrt{1-r^2}}} = \lim_{r\to 0} -\left(\frac{\sin^{-1}(r)}{r}\right)\sin^{-1}(r)\sqrt{1-r^2} = 0. \qquad (4.124)$$

The second integral in (4.122) has a singularity at $r = x$, which we remove to obtain

$$\begin{aligned}
\fint_{\frac{x}{2}}^{\frac{1+x}{2}} \frac{\log|r|}{\sqrt{1-r^2}(r-x)}\,dr &= \int_{\frac{x}{2}}^{\frac{1+x}{2}} \frac{\log|r|-\log|x|}{\sqrt{1-r^2}(r-x)}\,dr + \log|x| \fint_{\frac{x}{2}}^{\frac{1+x}{2}} \frac{dr}{\sqrt{1-r^2}(r-x)} \\
&= g_{2_C1c}(x) + \log|x|\left[\fint_{\frac{x}{2}}^{\frac{1+x}{2}}\left[\frac{1}{\sqrt{1-r^2}} - \frac{1}{\sqrt{1-x^2}}\right]\frac{dr}{r-x} + \frac{1}{\sqrt{1-x^2}}\fint_{\frac{x}{2}}^{\frac{1+x}{2}} \frac{dr}{r-x}\right] \\
&= g_{2_C1c}(x) + \frac{\log|x|}{\sqrt{1-x^2}}\int_{\frac{x}{2}}^{\frac{1+x}{2}} \frac{r+x}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\,dr \\
&\quad + \frac{\log|x|}{\sqrt{1-x^2}}\left[\log\left|\frac{1-x}{2}\right| - \log\left|\frac{x}{2}\right|\right] \\
&= g_{2_C1c}(x) + \frac{\log|x|}{\sqrt{1-x^2}}\Big[g_{2_C1d}(x) + \log|1-x| - \log|x|\Big]. \qquad (4.125)
\end{aligned}$$

In this case, the integral $g_{2\_C1c}(x)$ is continuous since

$$\lim_{r\to x}\frac{\log|r|-\log|x|}{r-x} = \lim_{r\to x}\frac{1}{r} = \frac{1}{x} < \infty, \quad \text{since } x \neq 0. \qquad (4.126)$$

We have removed all singularities from the integral $g_{2\_C1d}(x)$, so it is also continuous.

For the last integral of (4.122) we have

$$\begin{aligned}
&\fint_{\frac{1+x}{2}}^{1} \frac{\log|r|}{\sqrt{1-r^2}(r-x)}\,dr \\
&\quad= \frac{\sin^{-1}(r)\log|r|}{r-x}\Big|_{\frac{1+x}{2}}^{1} - \int_{\frac{1+x}{2}}^{1} \frac{\sin^{-1}(r)}{r(r-x)}\,dr + \int_{\frac{1+x}{2}}^{1} \frac{\sin^{-1}(r)\log|r|}{(r-x)^2}\,dr \\
&\quad= -\frac{2}{1-x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left|\frac{1+x}{2}\right| - g_{2_C1e}(x) + g_{2_C1f}(x), \qquad (4.127)
\end{aligned}$$

where the last two integrals no longer contain any singularities since $r > x > 0$.

Finally, combining (4.123), (4.125), and (4.127) in (4.122), we have for $x \in (0,1)$ and $q \in (-1,1)$

$$\begin{aligned}
g_2(x,q) &= g_2(x,0) \\
&= -\frac{2}{x}\sin^{-1}\left(\frac{x}{2}\right)\log\left|\frac{x}{2}\right| - \frac{2}{1-x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left|\frac{1+x}{2}\right| - g_{2_C1a}(x) \\
&\quad + g_{2_C1b}(x) + g_{2_C1c}(x) + \frac{\log|x|}{\sqrt{1-x^2}}\left[g_{2_C1d}(x) + \log|1-x| - \log|x|\right] \\
&\quad - g_{2_C1e}(x) + g_{2_C1f}(x). \qquad (4.128)
\end{aligned}$$

Case 3.2: $x \in [0,1),\ q = 1$, (i.e., $(x,q) \in T_2$)

By the symmetry property (4.118), this case also covers values of g_2 for $x \in (-1,0]$ and $q = -1$, i.e., $(x,q) \in B_1$. Isolating the singularities, we have

$$\begin{aligned}
g_2(x,1) &= \fint_{-1}^{1} \frac{\log|1-r|}{\sqrt{1-r^2}(r-x)}\,dr \\
&= \fint_{-1}^{-\frac{1+x}{2}} \frac{\log|1-r|}{\sqrt{1-r^2}(r-x)}\,dr + \int_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log|1-r|}{\sqrt{1-r^2}(r-x)}\,dr + \fint_{\frac{1+x}{2}}^{1} \frac{\log|1-r|}{\sqrt{1-r^2}(r-x)}\,dr, \qquad (4.129)
\end{aligned}$$

which we again work through piece by piece. For the first integral, we have

$$\begin{aligned}\fint_{-1}^{-\frac{1+x}{2}} \frac{\log|1-r|}{\sqrt{1-r^2}(r-x)}\,dr &= \frac{\sin^{-1}(r)\log|1-r|}{r-x}\Big|_{-1}^{-\frac{1+x}{2}} + \int_{-1}^{-\frac{1+x}{2}} \frac{\sin^{-1}(r)}{(1-r)(r-x)}\,dr \\ &\quad + \int_{-1}^{-\frac{1+x}{2}} \frac{\sin^{-1}(r)\log|1-r|}{(r-x)^2}\,dr \\ &= \frac{2}{1+3x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left|\frac{3+x}{2}\right| - \frac{\pi\log(2)}{2(1+x)} + g_{2\_C2a}(x) + g_{2\_C2b}(x),\end{aligned} \tag{4.130}$$

for which the singularity at $r=-1$ has been removed from both the remaining integrals $g_{2\_C2a}(x)$ and $g_{2\_C2b}(x)$.

In the second integral of (4.129), we remove the singularity at $r=x$ to obtain

$$\begin{aligned}&\fint_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log|1-r|}{\sqrt{1-r^2}(r-x)}\,dr \\ &= \int_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log|1-r|-\log|1-x|}{\sqrt{1-r^2}(r-x)}\,dr + \log|1-x|\fint_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{dr}{\sqrt{1-r^2}(r-x)} \qquad (4.131)\\ &= g_{2\_C2c}(x) + \log|1-x|\left[\fint_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \left(\frac{1}{\sqrt{1-r^2}} - \frac{1}{\sqrt{1-x^2}}\right)\frac{dr}{r-x}\right. \\ &\qquad\qquad \left. + \frac{1}{\sqrt{1-x^2}}\fint_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{dr}{r-x}\right] \\ &= g_{2\_C2c}(x) + \frac{\log|1-x|}{\sqrt{1-x^2}}\left[\int_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{r+x}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\,dr\right. \\ &\qquad\qquad \left. + \log\left|\frac{1-x}{2}\right| - \log\left|\frac{1+3x}{2}\right|\right] \\ &= g_{2\_C2c}(x) + \frac{\log|1-x|}{\sqrt{1-x^2}}\Big[g_{2\_C2d}(x) + \log|1-x| - \log|1+3x|\Big], \qquad (4.132)\end{aligned}$$

where by an argument similar to that of (4.126) it can be shown that the integrand of $g_{2\_C2c}(x)$ is continuous. The integral $g_{2\_C2d}(x)$ has no singularities. Moreover, we may reduce the complexity of this term by noticing that the denominator of the integrand

is an even function with respect to r and the limits of integration are symmetric. This yields

$$
\begin{aligned}
&g_{2\_C2d}(x) \\
&= \int_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{r}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\,dr + x\int_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{1}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\,dr \\
&= 0 + 2x\int_{0}^{\frac{1+x}{2}} \frac{1}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\,dr \\
&= 2x g^*_{2\_C2d}(x). \qquad (4.133)
\end{aligned}
$$

Thus we have for the second integral of (4.129)

$$
\begin{aligned}
&⨍_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log|1-r|}{\sqrt{1-r^2}(r-x)}\,dr \\
&\qquad = g_{2\_C2c}(x) + \frac{\log|1-x|}{\sqrt{1-x^2}}\Big[2x\, g^*_{2\_C2d}(x) + \log|1-x| - \log|1+3x|\Big]. \qquad (4.134)
\end{aligned}
$$

For the third integral of (4.129), we use the result in (4.112) to remove the singularity at $r = 1$. This yields

$$\begin{aligned}
&\fint_{\frac{1+x}{2}}^{1} \frac{\log|1-r|}{\sqrt{1-r^2}(r-x)}\,dr = -2\sqrt{1-r}\,\frac{\log|1-r|}{\sqrt{1+r}(r-x)}\Big|_{\frac{1+x}{2}}^{1} \\
&\quad + 2\fint_{\frac{1+x}{2}}^{1} \sqrt{1-r}\left[\frac{-1}{(1-r)\sqrt{1+r}(r-x)} - \frac{\log|1-r|}{2(1+r)^{3/2}(r-x)} - \frac{\log|1-r|}{\sqrt{1+r}(r-x)^2}\right] dr \\
&= \frac{2\sqrt{\frac{1-x}{2}}\log\left|\frac{1-x}{2}\right|}{\sqrt{\frac{3+x}{2}}\left(\frac{1-x}{2}\right)} - 2\fint_{\frac{1+x}{2}}^{1} \frac{dr}{\sqrt{1-r^2}(r-x)} \\
&\quad - \int_{\frac{1+x}{2}}^{1} \frac{\sqrt{1-r}\log|1-r|}{(1+r)^{3/2}(r-x)}\,dr - 2\int_{\frac{1+x}{2}}^{1} \frac{\sqrt{1-r}\log|1-r|}{\sqrt{1+r}(r-x)^2}\,dr \\
&= \frac{4\sqrt{1-x}\log\left|\frac{1-x}{2}\right|}{\sqrt{3+x}(1-x)} - 2\frac{\sin^{-1}(r)}{r-x}\Big|_{\frac{1+x}{2}}^{1} - 2\int_{\frac{1+x}{2}}^{1} \frac{\sin^{-1}(r)}{(r-x)^2}\,dr - g_{2_C2e}(x) - 2g_{2_C2f}(x) \\
&= \frac{4\log\left|\frac{1-x}{2}\right|}{\sqrt{3+x}\sqrt{1-x}} - \frac{\pi}{1-x} + \frac{4}{1-x}\sin^{-1}\left(\frac{1+x}{2}\right) - 2g_{2_C2g}(x) \\
&\quad - g_{2_C2e}(x) - 2g_{2_C2f}(x),
\end{aligned} \tag{4.135}$$

where the remaining integrals are all continuous.

Finally, applying (4.130), (4.134), and (4.135) to (4.129) yields, for all $x \in [0, 1)$,

$$\begin{aligned}
g_2(x,1) = {} & \frac{2}{1+3x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left|\frac{3+x}{2}\right| - \frac{\pi\log(2)}{2(1+x)} + \frac{4\log\left|\frac{1-x}{2}\right|}{\sqrt{3+x}\sqrt{1-x}} \\
& - \frac{1}{1-x}\left[\pi - 4\sin^{-1}\left(\frac{1+x}{2}\right)\right] \\
& + \frac{\log|1-x|}{\sqrt{1-x^2}}\left[2x\,g^*_{2_C2d}(x) + \log|1-x| - \log|1+3x|\right] \\
& + g_{2_C2a}(x) + g_{2_C2b}(x) + g_{2_C2c}(x) - g_{2_C2e}(x) - 2g_{2_C2f}(x) - 2g_{2_C2g}(x).
\end{aligned} \tag{4.136}$$

Case 3.3: $x \in [0,1),\ q = -1$ (i.e., $(x,q) \in B_2$)

By the symmetry property (4.118), we see that this case also covers values for $x \in (-1,0]$ and $q = 1$, i.e., $(x,q) \in T_1$. We again isolate singularities to obtain

$$\begin{aligned} g_2(x,-1) &:= \fint_{-1}^{1} \frac{\log|-1-r|}{\sqrt{1-r^2}(r-x)}\,dr = \fint_{-1}^{1} \frac{\log|1+r|}{\sqrt{1-r^2}(r-x)}\,dr \\ &= \fint_{-1}^{-\frac{1+x}{2}} \frac{\log|1+r|}{\sqrt{1-r^2}(r-x)}\,dr + \fint_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log|1+r|}{\sqrt{1-r^2}(r-x)}\,dr + \fint_{\frac{1+x}{2}}^{1} \frac{\log|1+r|}{\sqrt{1-r^2}(r-x)}\,dr. \end{aligned} \tag{4.137}$$

This case is very similar to Case 3.2 above. We will consider these integrals individually, but work from right to left to maintain a similar order to Case 3.2. We begin with the third integral above and remove the singularity at $r = 1$ using integration by parts. This yields

$$\begin{aligned} &\fint_{\frac{1+x}{2}}^{1} \frac{\log|1+r|}{\sqrt{1-r^2}(r-x)}\,dr \\ &\quad= \frac{\sin^{-1}(r)\log|1+r|}{r-x}\Big|_{\frac{1+x}{2}}^{1} - \int_{\frac{1+x}{2}}^{1} \frac{\sin^{-1}(r)}{(1+r)(r-x)}\,dr + \int_{\frac{1+x}{2}}^{1} \frac{\sin^{-1}(r)\log|1+r|}{(r-x)^2}\,dr \\ &\quad= \frac{\pi\log(2)}{2(1-x)} - \frac{2}{1-x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left|\frac{3+x}{2}\right| - g_{2\_C3a}(x) + g_{2\_C3b}(x), \end{aligned} \tag{4.138}$$

where these last two integrals are clearly continuous at $r = 1$.

In the second integral of (4.137), we remove the singularity at $r = x$ to obtain

$$\begin{aligned} &\fint_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log|1+r|}{\sqrt{1-r^2}(r-x)}\,dr \\ &\qquad= \int_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log|1+r| - \log|1+x|}{\sqrt{1-r^2}(r-x)}\,dr + \log|1+x|\fint_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{dr}{\sqrt{1-r^2}(r-x)}. \end{aligned}$$

The first integral on the right-hand side, which we denote $g_{2\_C3c}(x)$, is continuous by a computation similar to (4.126). The second integral is the same as that calculated in Case 3.2 in equation (4.131). Appealing to this result, and applying (4.133), yields

$$\begin{aligned}
&\fint_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log|1+r|}{\sqrt{1-r^2}(r-x)}\,dr \\
&\qquad = g_{2_C3c}(x) + \frac{\log|1+x|}{\sqrt{1-x^2}}\Big[2x\, g^*_{2_C2d}(x) + \log|1-x| - \log|1+3x|\Big]. \qquad (4.139)
\end{aligned}$$

For the first integral of (4.137), we use integration by parts and the result in (4.112) to remove the singularity at $r=-1$, which yields

$$\begin{aligned}
&\fint_{-1}^{-\frac{1+x}{2}} \frac{\log|1+r|}{\sqrt{1-r^2}(r-x)}\,dr \\
&= 2\sqrt{1+r}\frac{\log|1+r|}{\sqrt{1-r}(r-x)}\Big|_{-1}^{-\frac{1+x}{2}} \\
&\quad - 2\int_{-1}^{-\frac{1+x}{2}} \sqrt{1+r}\left[\frac{1}{(1+r)\sqrt{1-r}(r-x)} + \frac{\log|1+r|}{2(1-r)^{3/2}(r-x)} - \frac{\log|1+r|}{\sqrt{1-r}(r-x)^2}\right]dr \\
&= \frac{2\sqrt{\frac{1-x}{2}}\log\left|\frac{1-x}{2}\right|}{\sqrt{\frac{3+x}{2}}\left(\frac{-1-3x}{2}\right)} - 2\fint_{-1}^{-\frac{1+x}{2}} \frac{dr}{\sqrt{1-r^2}(r-x)} \\
&\quad - \int_{-1}^{-\frac{1+x}{2}} \frac{\sqrt{1+r}\log|1+r|}{(1-r)^{3/2}(r-x)}\,dr + 2\int_{-1}^{-\frac{1+x}{2}} \frac{\sqrt{1+r}\log|1+r|}{\sqrt{1-r}(r-x)^2}\,dr \\
&= -\frac{4\sqrt{1-x}\log\left|\frac{1-x}{2}\right|}{\sqrt{3+x}(1+3x)} - 2\frac{\sin^{-1}(r)}{r-x}\Big|_{-1}^{-\frac{1+x}{2}} - 2\int_{-1}^{-\frac{1+x}{2}} \frac{\sin^{-1}(r)}{(r-x)^2}\,dr \\
&\quad - g_{2_C3e}(x) + 2g_{2_C3f}(x) \\
&= -\frac{4\sqrt{1-x}\log\left|\frac{1-x}{2}\right|}{\sqrt{3+x}(1+3x)} - \frac{4\sin^{-1}\left(\frac{1+x}{2}\right)}{1+3x} + \frac{\pi}{1+x} \\
&\quad - 2g_{2_C3g}(x) - g_{2_C3e}(x) + 2g_{2_C3f}(x), \qquad (4.140)
\end{aligned}$$

where the remaining integrals are again continuous.

Finally, applying (4.138) – (4.140) to (4.137) yields, for all $x \in [0,1)$,

$$\begin{aligned} g_2(x,-1) = {} & \frac{\pi \log(2)}{2(1-x)} - \frac{2}{1-x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left|\frac{3+x}{2}\right| - \frac{4\sqrt{1-x}\log\left|\frac{1-x}{2}\right|}{\sqrt{3+x}(1+3x)} \\ & - \frac{4}{1+3x}\sin^{-1}\left(\frac{1+x}{2}\right) + \frac{\pi}{1+x} \\ & + \frac{\log|1+x|}{\sqrt{1-x^2}}\left[2x\, g^*_{2\_C2d}(x) + \log|1-x| - \log|1+3x|\right] \\ & - g_{2\_C3a}(x) + g_{2\_C3b}(x) + g_{2\_C3c}(x) - g_{2\_C3e}(x) + 2g_{2\_C3f}(x) - 2g_{2\_C3g}(x). \end{aligned} \tag{4.141}$$

Note that Cases 3.2 and 3.3 overlap at the points $(0, \pm 1)$, but by a change of variables, one can show that the value of $g_2(0,\pm 1)$ computed from the first case is equivalent to the value computed from the second.

Case 3.4: $x = 0,\ q \in (0,1)$ (i.e., $(x,q) \in M_1$)

Again we note that this case also covers values for $q \in (-1,0)$ by the symmetry property of g_2, i.e., $(x,q) \in M_2$. Isolating singularities, we have

$$\begin{aligned} & g_2(0,q) \\ & = \int_{-1}^{1} \frac{\log|q-r|}{r\sqrt{1-r^2}}\, dr \\ & = \int_{-1}^{-\frac{q}{2}} \frac{\log|q-r|}{r\sqrt{1-r^2}}\, dr + \int_{-\frac{q}{2}}^{\frac{q}{2}} \frac{\log|q-r|}{r\sqrt{1-r^2}}\, dr + \int_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{\log|q-r|}{r\sqrt{1-r^2}}\, dr + \int_{\frac{1+q}{2}}^{1} \frac{\log|q-r|}{r\sqrt{1-r^2}}\, dr, \end{aligned} \tag{4.142}$$

and we again look at each integral separately. For the first, integration by parts yields

$$\begin{aligned}
&\fint_{-1}^{-\frac{q}{2}} \frac{\log|q-r|}{r\sqrt{1-r^2}}\,dr \\
&\quad= \frac{\sin^{-1}(r)\log|q-r|}{r}\Big|_{-1}^{-\frac{q}{2}} - \int_{-1}^{-\frac{q}{2}} \sin^{-1}(r)\left[\frac{1}{r(r-q)} - \frac{\log|q-r|}{r^2}\right] dr \\
&\quad= \frac{\sin^{-1}(-\frac{q}{2})\log\left|\frac{3q}{2}\right|}{-\frac{q}{2}} - \frac{\sin^{-1}(-1)\log|q+1|}{-1} - \int_{-1}^{-\frac{q}{2}} \frac{\sin^{-1}(r)}{r(r-q)}\,dr \\
&\qquad+ \int_{-1}^{-\frac{q}{2}} \frac{\sin^{-1}(r)\log|q-r|}{r^2}\,dr \\
&\quad= \frac{2}{q}\sin^{-1}\left(\frac{q}{2}\right)\log\left|\frac{3q}{2}\right| - \frac{\pi}{2}\log|1+q| - g_{2_C4a}(q) + g_{2_C4b}(q). \qquad (4.143)
\end{aligned}$$

We add and subtract $\log|q|$ in the second integral of (4.142) to obtain

$$\begin{aligned}
\fint_{-\frac{q}{2}}^{\frac{q}{2}} \frac{\log|q-r|}{r\sqrt{1-r^2}}\,dr &= \int_{-\frac{q}{2}}^{\frac{q}{2}} \frac{\log|q-r| - \log|q|}{r\sqrt{1-r^2}}\,dr + \log|q| \fint_{-\frac{q}{2}}^{\frac{q}{2}} \frac{dr}{r\sqrt{1-r^2}} \\
&= \int_{-\frac{q}{2}}^{\frac{q}{2}} \frac{\log|q-r| - \log|q|}{r\sqrt{1-r^2}}\,dr \\
&= g_{2_C4c}(q), \qquad (4.144)
\end{aligned}$$

where we have used the fact that the second integrand above is an odd function, so its integral over symmetric limits is zero. The first integral above is continuous, since

$$\lim_{r\to 0} \frac{\log|q-r| - \log|q|}{r} = \lim_{r\to 0} \frac{1}{r-q} = -\frac{1}{q}, \qquad (4.145)$$

which is bounded for $q \in (0,1)$.

We remove the singularity at $r = q$ in the third integral of (4.142), which becomes

$$
\begin{aligned}
&\int_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{\log|q-r|}{r\sqrt{1-r^2}}\,dr \\
&\quad= \fint_{\frac{q}{2}}^{\frac{1+q}{2}} \left[\frac{1}{r}-\frac{1}{q}\right]\frac{\log|q-r|}{\sqrt{1-r^2}}\,dr + \frac{1}{q}\fint_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{\log|q-r|}{\sqrt{1-r^2}}\,dr \\
&\quad= \frac{1}{q}\int_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{(q-r)\log|q-r|}{r\sqrt{1-r^2}}\,dr + \frac{1}{q}\left[\sin^{-1}(r)\log|q-r|\,\Big|_{\frac{q}{2}}^{\frac{1+q}{2}} - \fint_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{\sin^{-1}(r)}{r-q}\,dr\right] \\
&\quad= \frac{1}{q}g_{2\_C4d}(q) + \frac{1}{q}\left[\sin^{-1}\left(\frac{1+q}{2}\right)\log\left|\frac{q-1}{2}\right| - \sin^{-1}\left(\frac{q}{2}\right)\log\left|\frac{q}{2}\right|\right] \\
&\qquad - \frac{1}{q}\left[\int_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{\sin^{-1}(r)-\sin^{-1}(q)}{r-q}\,dr + \sin^{-1}(q)\fint_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{dr}{r-q}\right] \\
&\quad= \frac{1}{q}g_{2\_C4d}(q) + \frac{1}{q}\sin^{-1}\left(\frac{1+q}{2}\right)\log\left|\frac{1-q}{2}\right| - \frac{1}{q}\sin^{-1}\left(\frac{q}{2}\right)\log\left|\frac{q}{2}\right| \\
&\qquad - \frac{1}{q}g_{2\_C4e}(q) - \frac{\sin^{-1}(q)}{q}\Big[\log|1-q| - \log|q|\Big].
\end{aligned}
\tag{4.146}
$$

The integral term $g_{2\_C4d}(q)$ on the right-hand side is continuous by (4.112). The second integral, $g_{2\_C4e}(q)$, is also continuous, since

$$
\lim_{r\to q}\frac{\sin^{-1}(r)-\sin^{-1}(q)}{r-q} = \frac{1}{\sqrt{1-q^2}} < \infty, \quad \text{for } q \in (0,1). \tag{4.147}
$$

The last integral on the right-hand side of (4.142) becomes

$$\begin{aligned}
&\fint_{\frac{1+q}{2}}^{1} \frac{\log|q-r|}{r\sqrt{1-r^2}}\,dr \\
&\quad= \frac{\sin^{-1}(r)\log|q-r|}{r}\Big|_{\frac{1+q}{2}}^{1} - \fint_{\frac{1+q}{2}}^{1} \sin^{-1}(r)\left[\frac{1}{r(r-q)} - \frac{\log|q-r|}{r^2}\right] dr \\
&\quad= \sin^{-1}(1)\log|q-1| - \frac{\sin^{-1}\left(\frac{1+q}{2}\right)\log\left|\frac{q-1}{2}\right|}{\frac{1+q}{2}} \\
&\qquad - \int_{\frac{1+q}{2}}^{1} \frac{\sin^{-1}(r)}{r(r-q)}\,dr + \int_{\frac{1+q}{2}}^{1} \frac{\sin^{-1}(r)\log|q-r|}{r^2}\,dr \\
&\quad= \frac{\pi}{2}\log|1-q| - \frac{2}{1+q}\sin^{-1}\left(\frac{1+q}{2}\right)\log\left|\frac{1-q}{2}\right| - g_{2_C4f}(x) + g_{2_C4g}(x).
\end{aligned} \tag{4.148}$$

Finally, we apply (4.143), (4.144), (4.146), and (4.148) to (4.142) to obtain

$$\begin{aligned}
g_2(0,q) = \frac{1}{q}\sin^{-1}\left(\frac{q}{2}\right)\left(2\log\left|\frac{3q}{2}\right| - \log\left|\frac{q}{2}\right|\right) - \frac{\pi}{2}\left(\log|1+q| - \log|1-q|\right) \\
+ \frac{1-q}{q(1+q)}\sin^{-1}\left(\frac{1+q}{2}\right)\log\left|\frac{1-q}{2}\right| - \frac{\sin^{-1}(q)}{q}\left[\log|1-q| - \log|q|\right] \\
- g_{2_C4a}(q) + g_{2_C4b}(q) + g_{2_C4c}(q) + \frac{1}{q}g_{2_C4d}(q) - \frac{1}{q}g_{2_C4e}(q) \\
- g_{2_C4f}(q) + g_{2_C4g}(q).
\end{aligned} \tag{4.149}$$

Case 3.5: $x = \pm 1,\ q \neq x$

To cancel the singularity at $r = x = \pm 1$, we use the coefficient $\sqrt{1-x^2}$ that occurs in Term 3. Thus we compute

$$g_2^*(x,q) := \sqrt{1-x^2}\,g_2(x,q). \tag{4.150}$$

Also, we have a similar symmetry property by applying (4.118), i.e.,

$$g_2^*(-x,-q) = \sqrt{1-(-x)^2}\, g_2(-x,-q) = -\sqrt{1-x^2}\, g_2(x,q) = -g_2^*(x,q). \qquad (4.151)$$

Case 3.5a: $x = 1,\ q = (-1,1)$ (i.e., $(x,q) \in R$)

By the symmetry property above, this case also covers values for $x = -1$, i.e., $(x,q) \in L$. First we isolate the singularities

$$\begin{aligned} g_2^*(1,q) &= \lim_{x\to 1} \sqrt{1-x^2} \fint_{-1}^{1} \frac{\log|q-r|}{\sqrt{1-r^2}(r-x)}\, dr \\ &= -\lim_{x\to 1} \sqrt{1-x^2} \left[\fint_{-1}^{\frac{1+q}{2}} \frac{\log|q-r|}{\sqrt{1-r^2}(x-r)}\, dr + \int_{\frac{1+q}{2}}^{1} \frac{\log|q-r|}{\sqrt{1-r^2}(x-r)}\, dr \right], \end{aligned} \qquad (4.152)$$

and compute each term. In the first term, we can directly substitute $x = 1$ into the integral and use integration by parts and our standard trick to remove the singularity.

This yields

$$
\begin{aligned}
&\fint_{-1}^{\frac{1+q}{2}} \frac{\log|q-r|}{\sqrt{1-r^2}(1-r)}\,dr \\
&\quad= \frac{\sqrt{1+r}}{\sqrt{1-r}}\log|q-r|\,\Big|_{-1}^{\frac{1+q}{2}} - \int_{-1}^{\frac{1+q}{2}} \frac{\sqrt{1+r}}{\sqrt{1-r}}\frac{dr}{r-q} \\
&\quad= \frac{\sqrt{\frac{3+q}{2}}\log\left|\frac{q-1}{2}\right|}{\sqrt{\frac{1-q}{2}}} - \int_{-1}^{\frac{1+q}{2}} \left[\frac{\sqrt{1+r}}{\sqrt{1-r}} - \frac{\sqrt{1+q}}{\sqrt{1-q}}\right]\frac{dr}{r-q} - \frac{\sqrt{1+q}}{\sqrt{1-q}}\int_{-1}^{\frac{1+q}{2}} \frac{dr}{r-q} \\
&\quad= \frac{\sqrt{3+q}\log\left|\frac{1-q}{2}\right|}{\sqrt{1-q}} - \frac{2}{\sqrt{1-q}}\int_{-1}^{\frac{1+q}{2}} \frac{dr}{\sqrt{1-r}(\sqrt{1+r}\sqrt{1-q}+\sqrt{1-r}\sqrt{1+q})} \\
&\qquad- \frac{\sqrt{1+q}}{\sqrt{1-q}}\log|r-q|\,\Big|_{-1}^{\frac{1+q}{2}} \\
&= \frac{\sqrt{3+q}\log\left|\frac{1-q}{2}\right|}{\sqrt{1-q}} - \frac{2}{\sqrt{1-q}}g_{2\_C5a\_1}(q) - \frac{\sqrt{1+q}}{\sqrt{1-q}}\left[\log\left|\frac{1-q}{2}\right| - \log|1+q|\right],
\end{aligned}
\tag{4.153}
$$

where the remaining integral $g_{2\_C5a\_1}(q)$ has no singularities. Therefore, multiplying this result by $\sqrt{1-x^2}$ and taking $x \to 1$, we see that the first term in (4.152) vanishes.

Thus the value of g_2 is given by the second term, which becomes

$$
\begin{aligned}
&g_2^*(1,q) \\
&\quad = -\lim_{x\to 1}\sqrt{1-x^2}\int_{\frac{1+q}{2}}^{1}\frac{\log|q-r|}{\sqrt{1-r^2}(x-r)}\,dr \\
&\quad = -\lim_{\varepsilon\to 0}\sqrt{1-(1-\varepsilon)^2}\int_{\frac{1+q}{2}}^{1-\varepsilon}\frac{\log|q-r|}{\sqrt{1-r^2}(1-r)}\,dr \\
&\quad = -\lim_{\varepsilon\to 0}\sqrt{\varepsilon}\sqrt{2-\varepsilon}\left[\frac{\sqrt{1+r}}{\sqrt{1-r}}\log|q-r|\,\Big|_{\frac{1+q}{2}}^{1-\varepsilon} - \int_{\frac{1+q}{2}}^{1-\varepsilon}\frac{\sqrt{1+r}}{\sqrt{1-r}}\frac{dr}{r-q}\right] \\
&\quad = -\lim_{\varepsilon\to 0}\sqrt{\varepsilon}\sqrt{2-\varepsilon}\left[\frac{\sqrt{2-\varepsilon}}{\sqrt{\varepsilon}}\log|q-1+\varepsilon| - \frac{\sqrt{\frac{3+q}{2}}}{\sqrt{\frac{1-q}{2}}}\log\left|\frac{q-1}{2}\right|\right. \\
&\qquad\qquad \left. - \int_{\frac{1+q}{2}}^{1-\varepsilon}\frac{1+r}{\sqrt{1-r^2}}\frac{dr}{r-q}\right] \\
&\quad = -2\log|q-1| + \lim_{\varepsilon\to 0}\sqrt{\varepsilon}\sqrt{2-\varepsilon}\left[\sin^{-1}(r)\frac{1+r}{r-q}\,\Big|_{\frac{1+q}{2}}^{1-\varepsilon}\right. \\
&\qquad \left. - \int_{\frac{1+q}{2}}^{1-\varepsilon}\sin^{-1}(r)\left(\frac{1}{r-q} - \frac{1+r}{(r-q)^2}\right)dr\right] \\
&\quad = -2\log|1-q| + \lim_{\varepsilon\to 0}\sqrt{\varepsilon}\sqrt{2-\varepsilon}\left[\sin^{-1}(1-\varepsilon)\frac{2-\varepsilon}{1-\varepsilon-q} - \sin^{-1}\left(\frac{1+q}{2}\right)\frac{3+q}{1-q}\right. \\
&\qquad \left. - \int_{\frac{1+q}{2}}^{1}\frac{\sin^{-1}(r)}{r-q}\,dr + \int_{\frac{1+q}{2}}^{1}\frac{\sin^{-1}(r)(1+r)}{(r-q)^2}\,dr\right] \\
&\quad = -2\log|1-q| + \lim_{\varepsilon\to 0}\sqrt{\varepsilon}\sqrt{2-\varepsilon}\left[-g_{2\_C5a\_2} + g_{2\_C5a\_3}\right] \\
&\quad = -2\log|1-q|\,, \qquad (4.154)
\end{aligned}
$$

since both remaining integrals are continuous and independent of ε.

Case 3.5b: $x = 1,\ q = -1$ (i.e., $(x, q) \in P_2$)

Again by the symmetry property in (4.151), this also covers values of g_2^* for $x = -1$ and $q = 1$, i.e., $(x, q) \in P_4$. Using the limit $x \to 1$, we have

$$\begin{aligned} g_2^*(1,-1) &= \lim_{x\to 1} \sqrt{1-x^2} \fint_{-1}^{1} \frac{\log|-1-r|}{\sqrt{1-r^2}(r-x)}\, dr \\ &= \lim_{\varepsilon\to 0} \sqrt{1-(1-\varepsilon)^2} \int_{-1+\varepsilon}^{1-\varepsilon} \frac{\log|1+r|}{\sqrt{1-r^2}(r-1)}\, dr \\ &= -\lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left[\frac{\sqrt{1+r}}{\sqrt{1-r}} \log|1+r| \Big|_{-1+\varepsilon}^{1-\varepsilon} - \int_{-1+\varepsilon}^{1-\varepsilon} \frac{\sqrt{1+r}}{\sqrt{1-r}} \frac{dr}{1+r} \right] \\ &= -\lim_{\varepsilon\to 0} \sqrt{\varepsilon}\sqrt{2-\varepsilon} \left[\frac{\sqrt{2-\varepsilon}}{\sqrt{\varepsilon}} \log|2-\varepsilon| - \frac{\sqrt{\varepsilon}}{\sqrt{2-\varepsilon}} \log|\varepsilon| - \sin^{-1}(r) \Big|_{-1+\varepsilon}^{1-\varepsilon} \right] \\ &= -2\log(2), \end{aligned} \tag{4.155}$$

since by (4.112)

$$\lim_{\varepsilon\to 0} \sqrt{\varepsilon} \log|\varepsilon| = 0. \tag{4.156}$$

Thus these two sub-cases may be combined to obtain

$$g_2^*(1,q) = -2\log|1-q|, \quad \forall\ q \in [-1,1). \tag{4.157}$$

A summary of the values of $g_2(x, q)$ is given in Table 4.4, with a corresponding summary of values for the entire Term 3 in Table 4.5. Finally, Table 4.6 lists all the continuous integrals appearing in the reformulation of the kernel in Tables 4.2 – 4.5.

Thus we have reformulated the kernel k to show that it is piecewise continuous everywhere on S except for a single logarithmic singularity in Term 1 over the subdomain D. In particular, k is square integrable on all of S and hence K is a Hilbert-Schmidt operator. This concludes the proof of Theorem 4.5. □

Table 4.4 Summary of the piecewise values of $g_2(x,q)$ on $S\backslash(P_1 \cup P_3 \cup D)$

Domain	$g_2(x,q)$
$I_3 \cup I_4$	$-\frac{2}{x}\sin^{-1}\left(\frac{x}{2}\right)\log\left\lvert\frac{x}{2}\right\rvert - \frac{2}{1-x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left\lvert\frac{1+x}{2}\right\rvert - g_{2\_C1a}(x)$ $+ g_{2\_C1b}(x) + g_{2\_C1c}(x) + \frac{\log\lvert x\rvert}{\sqrt{1-x^2}}\Big[g_{2\_C1d}(x) + \log\lvert 1-x\rvert - \log\lvert x\rvert\Big]$ $- g_{2\_C1e}(x) + g_{2\_C1f}(x)$
T_2	$\frac{2}{1+3x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left\lvert\frac{3+x}{2}\right\rvert - \frac{\pi\log(2)}{2(1+x)} + \frac{4\log\left\lvert\frac{1-x}{2}\right\rvert}{\sqrt{3+x}\sqrt{1-x}} - \frac{1}{1-x}\Big[\pi - 4\sin^{-1}\left(\frac{1+x}{2}\right)\Big]$ $+\frac{\log\lvert 1-x\rvert}{\sqrt{1-x^2}}\Big[2x\, g^*_{2\_C2d}(x) + \log\lvert 1-x\rvert - \log\lvert 1+3x\rvert\Big] + g_{2\_C2a}(x)$ $+g_{2\_C2b}(x) + g_{2\_C2c}(x) - g_{2\_C2e}(x) - 2g_{2\_C2f}(x) - 2g_{2\_C2g}(x)$
B_2	$\frac{\pi\log(2)}{2(1-x)} - \frac{2}{1-x}\sin^{-1}\left(\frac{1+x}{2}\right)\log\left\lvert\frac{3+x}{2}\right\rvert - \frac{4\sqrt{1-x}\log\left\lvert\frac{1-x}{2}\right\rvert}{\sqrt{3+x}(1+3x)}$ $-\frac{4}{1+3x}\sin^{-1}\left(\frac{1+x}{2}\right) + \frac{\pi}{1+x}$ $+\frac{\log\lvert 1+x\rvert}{\sqrt{1-x^2}}\Big[2x\, g^*_{2\_C2d}(x) + \log\lvert 1-x\rvert - \log\lvert 1+3x\rvert\Big] - g_{2\_C3a}(x)$ $+g_{2\_C3b}(x) + g_{2\_C3c}(x) - g_{2\_C3e}(x) + 2g_{2\_C3f}(x) - 2g_{2\_C3g}(x)$
M_1	$\frac{1}{q}\sin^{-1}\left(\frac{q}{2}\right)\left(2\log\left\lvert\frac{3q}{2}\right\rvert - \log\left\lvert\frac{q}{2}\right\rvert\right) - \frac{\pi}{2}\left(\log\lvert 1+q\rvert - \log\lvert 1-q\rvert\right)$ $+\frac{1-q}{q(1+q)}\sin^{-1}\left(\frac{1+q}{2}\right)\log\left\lvert\frac{1-q}{2}\right\rvert - \frac{\sin^{-1}(q)}{q}\Big[\log\lvert 1-q\rvert - \log\lvert q\rvert\Big]$ $-g_{2\_C4a}(q) + g_{2\_C4b}(q) + g_{2\_C4c}(q) + \frac{1}{q}g_{2\_C4d}(q) - \frac{1}{q}g_{2\_C4e}(q)$ $-g_{2\_C4f}(q) + g_{2\_C4g}(q)$
$P_2 \cup R$	$g^*_2(x,q) = \sqrt{1-x^2}\,g_2(x,q) = -2\log\lvert 1-q\rvert$

Table 4.5 Summary of the piecewise values of Term 3 of the kernel on $S = [-1,1]^2$

Subdomain	$T_3(x,q)$
$P_1 \cup P_3 \cup D$	$c_2\pi\log(2)\sqrt{1-x^2}$
$P_2 \cup R \cup T_2 \cup B_2 \cup M_1 \cup I_3 \cup I_4$	$c_2\pi\log(2)\sqrt{1-x^2} + c_2(q-x)g^*_2(x,q)$
$P_4 \cup L \cup T_1 \cup B_1 \cup M_2 \cup I_1 \cup I_2$	$c_2\pi\log(2)\sqrt{1-x^2} - c_2(q-x)g^*_2(-x,-q)$

Table 4.6 Summary of the continuous integrals appearing in the reformulation of k

Case	Domain	Integral
1.6	$M_1 \cup I_1 \cup I_3$	$\mathcal{I}_{1\_C6}(x,q) = \int_q^1 \frac{\sin^{-1}(r)}{(r-x)^2}\,dr$
1.7	$M_2 \cup I_2 \cup I_4$	$f_{1a}(x,q) = \int_q^1 \frac{dr}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})(\sqrt{1-r}\sqrt{1+x}+\sqrt{1-x}\sqrt{1+r})}$
		$f_{1b}(x,q) = \int_q^1 \frac{[r(x-1)-(x+3)]\,dr}{\sqrt{1+r}(\sqrt{2}+\sqrt{1+r})(\sqrt{1-r}(1+x)+\sqrt{1-x}(1+r))}$
1.8	D	$\mathcal{I}_{1\_C8a}(x,q) = \int_{\frac{1+q}{2}}^1 \frac{\sin^{-1}(r)}{(r-x)^2}\,dr$
		$\mathcal{I}_{1\_C8b}(x,q) = \int_q^{\frac{1+q}{2}} \frac{r+x}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\,dr$
3.1	$I_3 \cup I_4$	$g_{2\_C1a}(x) = \int_{-1}^{\frac{x}{2}} \frac{\sin^{-1}(r)}{r(r-x)}\,dr$
		$g_{2\_C1b}(x) = \int_{-1}^{\frac{x}{2}} \frac{\sin^{-1}(r)\log\lvert r\rvert}{(r-x)^2}\,dr$
		$g_{2\_C1c}(x) = \int_{\frac{x}{2}}^{\frac{1+x}{2}} \frac{\log\lvert r\rvert-\log\lvert x\rvert}{\sqrt{1-r^2}(r-x)}\,dr$
		$g_{2\_C1d}(x) = \int_{\frac{x}{2}}^{\frac{1+x}{2}} \frac{r+x}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\,dr$
		$g_{2\_C1e}(x) = \int_{\frac{1+x}{2}}^1 \frac{\sin^{-1}(r)}{r(r-x)}\,dr$
		$g_{2\_C1f}(x) = \int_{\frac{1+x}{2}}^1 \frac{\sin^{-1}(r)\log\lvert r\rvert}{(r-x)^2}\,dr$
3.2	T_2	$g_{2\_C2a}(x) = \int_{-1}^{-\frac{1+x}{2}} \frac{\sin^{-1}(r)}{(1-r)(r-x)}\,dr$
		$g_{2\_C2b}(x) = \int_{-1}^{-\frac{1+x}{2}} \frac{\sin^{-1}(r)\log\lvert 1-r\rvert}{(r-x)^2}\,dr$
		$g_{2\_C2c}(x) = \int_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log\lvert 1-r\rvert-\log\lvert 1-x\rvert}{\sqrt{1-r^2}(r-x)}\,dr$
		$g^*_{2\_C2d}(x) = \int_0^{\frac{1+x}{2}} \frac{1}{\sqrt{1-r^2}(\sqrt{1-x^2}+\sqrt{1-r^2})}\,dr$
		$g_{2\_C2e}(x) = \int_{\frac{1+x}{2}}^1 \frac{\sqrt{1-r}\log\lvert 1-r\rvert}{(1+r)^{3/2}(r-x)}\,dr$
		$g_{2\_C2f}(x) = \int_{\frac{1+x}{2}}^1 \frac{\sqrt{1-r}\log\lvert 1-r\rvert}{\sqrt{1+r}(r-x)^2}\,dr$

Table 4.6 Continued

Case	Domain	Integral
		$g_{2\_C2g}(x) = \int_{\frac{1+x}{2}}^{1} \frac{\sin^{-1}(r)}{(r-x)^2}\, dr$
3.3	B_2	$g_{2\_C3a}(x) = \int_{\frac{1+x}{2}}^{1} \frac{\sin^{-1}(r)}{(1+r)(r-x)}\, dr$
		$g_{2\_C3b}(x) = \int_{\frac{1+x}{2}}^{1} \frac{\sin^{-1}(r)\log\lvert 1+r\rvert}{(r-x)^2}\, dr$
		$g_{2\_C3c}(x) = \int_{-\frac{1+x}{2}}^{\frac{1+x}{2}} \frac{\log\lvert 1+r\rvert - \log\lvert 1+x\rvert}{\sqrt{1-r^2}(r-x)}\, dr$
		$g_{2\_C3e}(x) = \int_{-1}^{-\frac{1+x}{2}} \frac{\sqrt{1+r}\log\lvert 1+r\rvert}{(1-r)^{3/2}(r-x)}\, dr$
		$g_{2\_C3f}(x) = \int_{-1}^{-\frac{1+x}{2}} \frac{\sqrt{1+r}\log\lvert 1+r\rvert}{\sqrt{1-r}(r-x)^2}\, dr$
		$g_{2\_C3g}(x) = \int_{-1}^{-\frac{1+x}{2}} \frac{\sin^{-1}(r)}{(r-x)^2}\, dr$
3.4	M_1	$g_{2\_C4a}(q) = \int_{-1}^{-\frac{q}{2}} \frac{\sin^{-1}(r)}{r(r-q)}\, dr$
		$g_{2\_C4b}(q) = \int_{-1}^{-\frac{q}{2}} \frac{\sin^{-1}(r)\log\lvert q-r\rvert}{r^2}\, dr$
		$g_{2\_C4c}(q) = \int_{-\frac{q}{2}}^{\frac{q}{2}} \frac{\log\lvert q-r\rvert - \log\lvert q\rvert}{r\sqrt{1-r^2}}\, dr$
		$g_{2\_C4d}(q) = \int_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{(q-r)\log\lvert q-r\rvert}{r\sqrt{1-r^2}}\, dr$
		$g_{2\_C4e}(q) = \int_{\frac{q}{2}}^{\frac{1+q}{2}} \frac{\sin^{-1}(r) - \sin^{-1}(q)}{r-q}\, dr$
		$g_{2\_C4f}(q) = \int_{\frac{1+q}{2}}^{1} \frac{\sin^{-1}(r)}{r(r-q)}\, dr$
		$g_{2\_C4g}(q) = \int_{\frac{1+q}{2}}^{1} \frac{\sin^{-1}(r)\log\lvert q-r\rvert}{r^2}\, dr$

4.3 Algorithm for Estimating the L^2 Norm of the Kernel

In the previous section, we showed that for any subdomain given in Table 4.1, the value of the kernel on that domain may be computed from (4.52) using the piecewise values given in Tables 4.2 – 4.5. We use this reformulation of the original kernel k (4.47) to construct an algorithm for estimating the L^2 norm of k. As described in the discussion preceding Theorem 4.5, this algorithm yields a threshold value for the surface tension parameter γ_1 given any fixed parameter γ_0, in accordance with (4.51).

The algorithm takes advantage of the fact that we have reduced the kernel to piecewise terms that are continuous everywhere on S except for a logarithmic singularity on D. Since we compute the L^2 integral of the kernel, we can easily avoid this logarithmically singular term by noting that, in $\mathbb{R}^2$, the diagonal D is a set of measure zero, so it is sufficient to compute the norm on the set $S \backslash D$. Equivalently, we may take the norm over all of S simply by setting the value of the kernel to be identically zero on the diagonal. With this convention, we can evaluate the kernel k everywhere on S. We compute the continuous integral terms that appear in the reformulation of k, as listed in Table 4.6, using Gaussian quadrature. The rest of the terms in the kernel are simple continuous functions that may be evaluated directly.

We estimate the L^2 norm of the kernel using its cubic spline interpolant. We used Mathematica® (Wolfram Research, Inc. 2010) to construct the algorithm, with the major steps summarized by:

1. Set the values of Poisson's ratio ν and the first surface tension parameter γ_0.

2. Define a uniform set of nodes on S.

3. Compute the value of $k(x, q)$ at these nodes. We use the reformulation of the kernel as given in Section 4.2 to determine these values, applying Gaussian quadra-

ture to compute all of the continuous integrals defined in Table 4.6. We set k to be zero on the diagonal D.

4. Construct a cubic spline interpolant for the square of these values on the given nodes.

5. Compute the double integral of this interpolant over S.

6. Finally, take the square root of the result to obtain an estimate for the L^2 norm of the kernel.

The code itself may be found in Appendix D.

4.4 Numerical Results for the L^2 Norm of k

The Mathematica® program we constructed evaluates the kernel, finds the cubic spline interpolant of the square of the kernel, and computes its double integral over S to find an approximation to the L^2 norm of k. We denote this value by γ_1^{min} since it serves as a lower bound for $|\gamma_1|$. We note that the algorithm is dependent only on ν and γ_0, and in particular, is independent of the loading σ. As mentioned in Section 3.2, we assume an idealized brittle material similar to silicon and hence use a corresponding value for Poisson's ratio, i.e., $\nu = 0.2315$.

To determine the number of nodes required to obtain an accurate computation of the L^2 norm, we first fixed $\gamma_0 = 0$ and executed the Mathematica® program for an increasing number of nodes. The results are given in Table 4.7, where `nNodes` is the number of nodes along the interval $(0, 1)$ and `tNodes` is the total number of interpolation nodes in S. The step size is the distance between consecutive nodes. We also show the total time required for the computation and the difference between consecutive L^2 norms. We find that the value of γ_1^{min} converges to approximately 1.425 as the number of nodes is increased. However, as the total number of nodes

Table 4.7 L^2 norm of the kernel (γ_1^{min}) for an increasing number of nodes with fixed $\gamma_0 = 0$

nNodes	tNodes	Step size	γ_1^{min}	Time (h:m:s)	Difference
25	2,601	0.04	1.37051	00:02:42	—
50	10,201	0.02	1.39923	00:10:49	0.02872
75	22,801	0.0133333	1.40879	00:24:20	0.00956
100	40,401	0.01	1.41358	00:43:19	0.00479
125	63,001	0.008	1.41644	01:08:35	0.00286
150	90,601	0.00666667	1.41836	01:40:08	0.00192
175	123,201	0.00571429	1.41972	02:09:45	0.00136
200	160,801	0.005	1.42075	03:27:06	0.00103
225	203,401	0.00444444	1.42154	03:43:35	0.00079
250	251,001	0.004	1.42218	05:15:14	0.00064
275	303,601	0.00363636	1.4227	06:16:23	0.00052
300	361,201	0.00333333	1.42314	06:29:25	0.00044
325	423,801	0.00307692	1.42351	08:09:40	0.00037
350	491,401	0.00285714	1.42382	09:51:18	0.00031
375	564,001	0.00266667	1.4241	10:19:37	0.00028
400	641,601	0.0025	1.42433	12:30:15	0.00023
425	724,201	0.00235294	1.42609	13:27:16	0.00176
450	811,801	0.00222222	1.42473	15:18:50	−0.00136
475	904,401	0.00210526	Memory exceeded		
500	1,002,001	0.002	Memory exceeded		

approaches one million, the run time exceeds 10 hours and Mathematica® runs out of memory.

One of the main questions that is raised is what value should be assigned to the parameter γ_0. This parameter corresponds to the surface tension of a material interface in the absence of curvature. One of the goals for our future work will be to determine the value of γ_0 experimentally. For now, we simply compute the L^2 norm for a range

Table 4.8 L^2 norm of the kernel (γ_1^{min}) for various values of γ_0 for fixed `nNodes` = 300

γ_0	γ_1^{min}	γ_1^{min}/γ_0
0	1.42314	—
0.00001	1.42316	142,316
0.0001	1.42337	14,233.7
0.001	1.42544	1,425.44
0.01	1.44629	144.629
0.1	1.66267	16.6267
1	4.11187	4.11187
10	29.9987	2.99987
100	289.635	2.89635
1,000	2886.11	2.88611
10,000	28850.84	2.88508
100,000	288498.18	2.88498
1,000,000	2884971.56	2.88497

of values of γ_0. We note from Table 4.7 that we have fairly good accuracy without requiring an excessive run time when we choose `nNodes` = 300. Fixing this value, we ran the program for various values of γ_0, the results of which are shown in Table 4.8. We see that γ_1^{min} converges to 1.42314 as γ_0 decreases to zero. We also compute the ratio of γ_1^{min} to γ_0 and find that this ratio converges to approximately 2.885 as γ_0 increases.

In summary, we have shown an alternative proof that the Sendova-Walton model with curvature-dependent surface tension, in the absence of body forces, yields bounded crack-tip stresses and strains for all but a countable number of values of γ_1, given any fixed γ_0. In particular, the proof made it possible to construct an algorithm to obtain a threshold value for γ_1. In other words, with this algorithm we have the

ability to produce pairs of surface tension parameters that guarantee bounded crack-tip stresses. This will be critical information to include in the forthcoming numerical implementation of this model.

5. CONCLUSIONS AND FUTURE WORK

We have developed a numerical model of brittle fracture using the finite element method. The model is an application of the Sendova-Walton fracture theory with constant surface tension to the classical Griffith crack problem. We have shown that the numerical model agrees with the theory, predicting a logarithmic singularity in the crack-tip stress and a finite angle opening profile. The results also exhibit additional expected trends, namely, that the deformation increases with the far-field loading and decreases as the surface tension increases.

One of the immediate advantages of this model is that it allows for a fracture criterion based on the crack-tip opening angle, or equivalently, the opening displacement. This is only possible because the predicted angle is finite (i.e., an acute angle with finite slope), unlike the infinite opening angle (i.e., a right angle with infinite slope) predicted by LEFM. We plan to compare our results with experimental data, from which we can determine the maximum opening angle required for fracture. However, one potential disadvantage to this model is that it does not completely remove the singularity in the crack-tip stress. As yet, it is unclear how significant a role this singularity plays in the model, since it has been reduced from the square root singularity predicted by LEFM to a logarithmic singularity. Consequently, the stress singularity may not be apparent in the data until the finite elements that cover the crack tip are sufficiently small.

Such elements may be necessary for many small-scale applications, and therefore we are still very interested in developing the numerical model for the case of curvature-dependent surface tension. We have shown an alternative proof that the Sendova-Walton theory yields bounded crack-tip stresses when taking zero body force and

an appropriate pair of parameters in the curvature-dependent surface tension. In particular, the algorithm we presented in Section 4.3 shows how to obtain such a pair of bounding surface tension parameters. With this tool in hand, and the constant surface tension model as a basis, we have developed a solid foundation for implementing the curvature-dependent model. In addition to removing the crack-tip stress singularity, this model also yields a cusp-like opening profile, thereby correcting the two main inconsistencies of the LEFM model. Another advantage of this model is that it will provide a fracture criterion based on maximum crack-tip stress, again only possible since the predicted stress is finite.

Our initial investigation into the implementation of the curvature-dependent model has been hindered by the difficulties arising from attempting to use the FEM on the resulting weak formulation. Recall from Section 4 that the (linearized) curvature-dependent surface tension is given by

$$\tilde{\gamma}(x) = \gamma_0 + \gamma_1 u_{2,11}(x,0) + \text{h.o.t}, \tag{5.1}$$

with corresponding linearized JMB equations

$$\begin{aligned} \sigma_{12}(x,0) &= -\gamma_1 u_{2,111}(x,0) + \text{h.o.t} \\ \sigma_{22}(x,0) &= -\gamma_0 u_{2,11}(x,0) + \text{h.o.t} \end{aligned}, \text{ for } |x| \leq 1. \tag{5.2}$$

Similarly to the construction of the variational problem in Definition 2, we apply the JMB above as a boundary condition on the crack surface to obtain the weak formulation for the curvature-dependent model over the finite computational square domain Q (see Figure 3.1)

$$a(\mathbf{u}_\kappa, \mathbf{v}) - \int_{\Gamma_C} \gamma_0 v_2 u_{2,11} - \int_{\Gamma_C} \gamma_1 v_1 u_{2,111} = (\mathbf{v}, \mathbf{b}_\kappa)_\Omega + \int_{\Gamma_T} \sigma v_2, \tag{5.3}$$

where $a(\cdot,\cdot)$ is the same bilinear form as before, given in (3.4).

The challenge in applying the FEM to this weak formulation occurs in the third-derivative term. Any solution requires $u_2 \in H^3(\Gamma_C)$, far smoother than the other terms. Even after integrating by parts, we would need $u_2 \in H^2(\Gamma_C)$. However, this is not necessarily an advantage, since the boundary terms that appear after integrating do not vanish, and the second derivative still requires more smoothness over the crack surface than the rest of the terms in the formulation.

We have considered a number of different approaches to resolve this issue. One idea is to convert everything to the deformed configuration, using the fully nonlinear expression for the curvature-dependent surface tension. The advantage to this approach is that the second-derivative term in (5.3) turns into a term involving the full mean curvature, which may be computed from a strictly geometric calculation on the mesh. Then the third derivative in (5.3) turns into the gradient of curvature. However, the geometric calculation of curvature does not guarantee continuity, so the gradient may not exist.

Another approach would be to use non-standard finite elements that have additional global smoothness. In particular, we looked at the second-order Hermite Bogner-Fox-Schmit rectangular element (Bogner, Fox, and Schmit 1965) which is globally C^2. However, these elements have a number of disadvantages, including requiring a very large number of degrees of freedom and permitting only uniform rectangular meshes. This severely limited our ability to obtain accurate results.

As a consequence, we conclude that the FEM may not be the best method to use to implement the model in the case of curvature-dependent surface tension. Some alternatives we are considering are spectral methods and mesh-less B-spline algorithms. We plan to develop the curvature-dependent model using a B-spline code, since it will provide plenty of smoothness to capture the behavior of the third-derivative term.

In addition, both the constant and curvature-dependent surface tension models have yet to be validated experimentally. To our knowledge, there is no readily available empirical data from experiments corresponding to our problem that we could use for validation. Instead, we are currently working to develop appropriate experiments that will yield the necessary data for validation. In particular, we would like to be able to determine the value of γ_0 for a given material for the constant surface tension model.

There are many other opportunities for future projects motivated by this research that seek to generalize the model for application to more complex fracture problems. One direction we are looking at currently is modifying the Sendova-Walton theory for mode-II, mode-III, and mixed-mode cracks. Other avenues include allowing time-dependent crack propagation, adding nonzero mutual body forces, and working with various loading conditions and geometries. Finally, we would like to expand the theory for different types of materials, including heterogeneous, anisotropic, and functionally graded materials, or problems involving two bonded materials with cracks along the bonding plane.

REFERENCES

Abraham FF (2001) The atomic dynamics of fracture. J Mech Phys Solids 49:2095–2111

Abraham FF, Brodbeck D, Rudge WE, Xu XP (1997) A molecular dynamics investigation of rapid fracture mechanics. J Mech Phys Solids 45:1595–1619

Bangerth W, Hartmann R, Kanschat G (2007) deal.II – A general-purpose object-oriented finite element library. ACM Trans Math Softw 33:24/1–24/27

Bangerth W, Hartmann R, Kanschat G (2012) `deal.II` Differential Equations Analysis Library, Technical Reference. `http://www.dealii.org`

Belytschko T, Xiao SP (2004) Multiscale analysis with atomistic/continuum models for fracture. In: Yao, ZH and Yuan, MW and Zhong, WX (ed) Computational Mechanics, Proceedings of the 6th World Congress on Computational Mechanics, Beijing, China pp 1–9

Bogner F, Fox R, Schmit L (1965) The generation of inter-element-compatible stiffness and mass matrices by the use of interpolation formulas. In: Proceedings of the Conference on Matrix Methods in Structural Mechanics. Wright Patterson A.F.B., Dayton, OH, pp 397–443

Bourdin B, Francfort GA, Marigo J-J (2008) The variational approach to fracture. J Elasticity 91:5–148

Broberg KB (1999) Cracks and Fracture. Academic Press, San Diego, California

Burke S, Ortner C, Süli E (2010) An adaptive finite element approximation of a variational model of brittle fracture. SIAM J Numer Anal 48:980–1012

Curtin WA, Miller RE (2003) Atomistic/continuum coupling in computational materials science. Model Simul Mater Sc 11:R33–R68

Dal Maso G, Francfort GA, Toader R (2005) Quasistatic crack growth in nonlinear elasticity. Arch Ration Mech An 176:165–225

Davis JR (ed) (2004) Tensile Testing, 2nd edn. ASM International, Materials Park, OH

Edmunds DE, Evans WD (1987) Spectral Theory and Differential Operators. Clarendon Press, Oxford

Erdogan F (2000) Fracture mechanics. Int J Solids Struct 37:171–183

Ern A, Guermond J-L (2004) Theory and Practice of Finite Elements. Springer-Verlag, New York

Francfort GA, Garroni A (2006) A variational view of partial brittle damage evolution. Arch Ration Mech An 182:125–152

Francfort GA, Larsen CJ (2003) Existence and convergence for quasi-static evolution in brittle fracture. Commun Pur Appl Math 56:1465–1500

Francfort GA, Marigo J-J (1998) Revisiting brittle fracture as an energy minimization problem. J Mech Phys Solids 46:1319–1342

Gibbs JW (1928) The Collected Works of J. Willard Gibbs, vol I – Thermodynamics. Longmans, Green, and Co., New York

Griffith AA (1921) The phenomena of rupture and flow in solids. Philos T R Soc Lond A221:163–198

Grossmann C, Roos H-G, Stynes M (2007) Numerical Treatment of Partial Differential Equations. Springer-Verlag, Berlin

Gurtin ME (2003) An Introduction to Continuum Mechanics. Academic Press, San Diego, California

Ioffe Physico-Technical Institute (2001) New Semiconductor Materials. Characteristics and Properties. Si - Silicon. `http://www.ioffe.ru/SVA/NSM/Semicond/Si/mechanic.html`, Accessed 6 February 2012

Irwin GR (1948) Fracture dynamics. In: Proceedings of the ASM Symposium on Fracturing of Metals, pp 147–166

Jiang S, Rokhlin V (2003) Second kind integral equations for the classical potential theory on open surfaces I: analytical apparatus. J Comput Phys 191:40–74

Keener JP (2000) Principles of Applied Mathematics: Transformation and Approximation, Revised edn. Westview Press, Cambridge, MA

The MathWorks, Inc (2012) MATLAB®, version R2012b. Natick, MA

Miller R, Tadmor EB, Phillips R, Ortiz M (1998) Quasicontinuum simulation of fracture at the atomic scale. Model Simul Mater Sc 6:607–638

Miller RE, Tadmor EB (2007) Hybrid continuum mechanics and atomistic methods for simulating materials deformation and failure. MRS Bull 32:920–926

Muskhelishvili NI (1977) Singular Integral Equations. Noordhoff International Publishing, Leyden, The Netherlands

Oh E-S, Walton JR, Slattery JC (2006) A theory of fracture based upon an extension of continuum mechanics to the nanoscale. J Appl Mech - T ASME 73:792–798

Sadd MH (2009) Elasticity: Theory, Applications, and Numerics, 2nd edn. Elsevier, Amsterdam, The Netherlands

Sendova T, Walton JR (2010) A new approach to the modeling and analysis of fracture through extension of continuum mechanics to the nanoscale. Math Mech Solids 15:368–413

Sikora L (2012) Properties of silicon and silicon wafers. EL-CAT Inc. `http://www.el-cat.com/silicon-properties.htm`, Accessed 6 February 2012

Slattery J, Oh E-S, Fu K (2004) Extension of continuum mechanics to the nanoscale. Chem Eng Sci 59:4621–4635

Slattery JC, Sagis L, Oh E-S (2007) Interfacial Transport Phenomena, 2nd edn. Springer, New York, NY

Tadmor EB, Phillips R, Ortiz M (1996) Mixed atomistic and continuum models of deformation in solids. Langmuir 12:4529–4534

Vandenberghe S (1999) Silicon Mechanical Properties. `http://inmmc.org/ftp/material/silicon-mechanical.html`, Accessed 6 February 2012

Wolfram Research, Inc (2010) Mathematica®, version 8.0. Champaign, IL

APPENDIX A

COMPONENT JUMP MOMENTUM BALANCE DERIVATION

We re-derive the component form equations of the jump momentum balance (JMB) on the upper crack surface Σ^+ to correct a slight error in (Sendova and Walton 2010, Appendix B). Recall the JMB equation (2.3)

$$J(\mathrm{div}_{(\sigma)}\,\mathbf{T}^{(\sigma)}\otimes\mathbf{n}^-)_m\mathbf{F}^{-T}\mathbf{N}^- + [\![\mathbf{T}_\kappa]\!]\,\mathbf{N}^- = 0, \quad \text{on } \Sigma^+. \tag{A.1}$$

To find the corresponding component form, we employ the same definitions as in (Sendova and Walton 2010), namely,

$$\begin{aligned}
\mathbf{n}^- &= \frac{1}{\sqrt{(1+u_{1,1})^2+u_{2,1}^2}}\langle -u_{2,1}, 1+u_{1,1}\rangle^{\mathsf{T}},\\
\mathbf{N}^- &= \langle 0,1\rangle^{\mathsf{T}}, \quad \text{and}\\
\mathbf{F} &= \begin{pmatrix} 1+u_{1,1} & u_{1,2} \\ u_{2,1} & 1+u_{2,2} \end{pmatrix}.
\end{aligned} \tag{A.2}$$

Applying the definition of the tensor product and using the component form of $\mathbf{T}_\kappa$, they have

$$(J\mathbf{F}^{-T}\mathbf{N}^-\cdot\mathbf{n}^-)\,\mathrm{div}_{(\sigma)}\,\mathbf{T}^{(\sigma)} + \begin{pmatrix}\sigma_{12}\\ \sigma_{22}\end{pmatrix} = 0. \tag{A.3}$$

They also correctly compute the divergence term as

$$\begin{aligned}\operatorname{div}_{(\sigma)}\mathbf{T}^{(\sigma)} &= \operatorname{grad}_{(\sigma)}\tilde{\gamma} - \tilde{\gamma}\mathbf{n}^{-}\operatorname{div}_{(\sigma)}\mathbf{n}^{-} \\ &= \frac{\tilde{\gamma}'(X_1)}{(1+u_{1,1})^2+u_{2,1}^2}\langle(1+u_{1,1})^2,(1+u_{1,1})u_{2,1}\rangle^{\mathsf{T}} \\ &\quad - \tilde{\gamma}\mathbf{n}^{-}\frac{u_{2,1}^2u_{1,12}+u_{2,1}(1+u_{1,1})(u_{1,11}-u_{2,12})-(1+u_{1,1})^2u_{2,11}}{((1+u_{1,1})^2+u_{2,1}^2)^{3/2}}. \quad \text{(A.4)}\end{aligned}$$

The main correction we make is in the computation of $J\mathbf{F}^{-T}\mathbf{N}^{-}\cdot\mathbf{n}^{-}$ (cf. Sendova and Walton 2010, equation (86)). To compute this term, we note that

$$\mathbf{F}^{-1} = \frac{\mathrm{Adj}\mathbf{F}}{\det\mathbf{F}}, \quad \text{(A.5)}$$

where

$$\mathrm{Adj}\mathbf{F} = \begin{pmatrix} 1+u_{2,2} & -u_{1,2} \\ -u_{2,1} & 1+u_{1,1} \end{pmatrix}. \quad \text{(A.6)}$$

Then we have

$$\begin{aligned}J\mathbf{F}^{-\mathsf{T}}\mathbf{N}^{-}\cdot\mathbf{n}^{-} &= \det\mathbf{F}\left(\frac{\mathrm{Adj}\mathbf{F}}{\det\mathbf{F}}\right)^{\mathsf{T}}\mathbf{N}^{-}\cdot\mathbf{n}^{-} \\ &= (\mathrm{Adj}\mathbf{F})^{\mathsf{T}}\mathbf{N}^{-}\cdot\mathbf{n}^{-} \\ &= \begin{pmatrix} -u_{2,1} \\ 1+u_{1,1} \end{pmatrix}\cdot\frac{1}{\sqrt{(1+u_{1,1})^2+u_{2,1}^2}}\begin{pmatrix} -u_{2,1} \\ 1+u_{1,1} \end{pmatrix} \\ &= \frac{1}{\sqrt{(1+u_{1,1})^2+u_{2,1}^2}}\left[u_{2,1}^2+(1+u_{1,1})^2\right] \\ &= \sqrt{(1+u_{1,1})^2+u_{2,1}^2}. \quad \text{(A.7)}\end{aligned}$$

Combining this result with (A.3) and (A.4) yields the corrected component form of the jump momentum balance on Σ^+, given by

$$\sigma_{12} = -\sqrt{(1+u_{1,1})^2+u_{2,1}^2}\left(\frac{\tilde{\gamma}'(X_1)(1+u_{1,1})^2}{(1+u_{1,1})^2+u_{2,1}^2}\right.$$
$$\left. + \frac{\tilde{\gamma}(X_1)u_{2,1}\left[u_{2,1}^2u_{1,12}+u_{2,1}(1+u_{1,1})(u_{1,11}-u_{2,12})-(1+u_{1,1})^2u_{2,11}\right]}{\left((1+u_{1,1})^2+u_{2,1}^2\right)^2}\right), \quad \text{(A.8)}$$

$$\sigma_{22} = -\sqrt{(1+u_{1,1})^2+u_{2,1}^2}\left(\frac{\tilde{\gamma}'(X_1)(1+u_{1,1})u_{2,1}}{(1+u_{1,1})^2+u_{2,1}^2}\right.$$
$$\left. - \frac{\tilde{\gamma}(X_1)(1+u_{1,1})\left[u_{2,1}^2u_{1,12}+u_{2,1}(1+u_{1,1})(u_{1,11}-u_{2,12})-(1+u_{1,1})^2u_{2,11}\right]}{\left((1+u_{1,1})^2+u_{2,1}^2\right)^2}\right). \quad \text{(A.9)}$$

An analogous computation over Σ^-, where

$$\mathbf{n}^+ = \frac{1}{\sqrt{(1+u_{1,1})^2+u_{2,1}^2}}\langle -u_{2,1}, -(1+u_{1,1})\rangle^{\mathsf{T}}, \quad \text{and} \quad \text{(A.10)}$$

$$\mathbf{N}^+ = \langle 0, -1\rangle^{\mathsf{T}}, \quad \text{(A.11)}$$

yields the component JMB on the lower surface

$$\sigma_{12} = -\frac{(1+u_{1,1})^2 - u_{2,1}^2}{\sqrt{(1+u_{1,1})^2 + u_{2,1}^2}} \left(\frac{\tilde{\gamma}'(X_1)(1+u_{1,1})^2}{(1+u_{1,1})^2 + u_{2,1}^2} + \frac{\tilde{\gamma}(X_1) u_{2,1} \left[-u_{2,1}^2 u_{1,12} + u_{2,1}(1+u_{1,1})(u_{1,11}+u_{2,12}) - (1+u_{1,1})^2 u_{2,11} \right]}{\left((1+u_{1,1})^2 + u_{2,1}^2 \right)^2} \right), \tag{A.12}$$

$$\sigma_{22} = -\frac{(1+u_{1,1})^2 - u_{2,1}^2}{\sqrt{(1+u_{1,1})^2 + u_{2,1}^2}} \left(\frac{-\tilde{\gamma}'(X_1)(1+u_{1,1}) u_{2,1}}{(1+u_{1,1})^2 + u_{2,1}^2} + \frac{\tilde{\gamma}(X_1)(1+u_{1,1}) \left[-u_{2,1}^2 u_{1,12} + u_{2,1}(1+u_{1,1})(u_{1,11}+u_{2,12}) - (1+u_{1,1})^2 u_{2,11} \right]}{\left((1+u_{1,1})^2 + u_{2,1}^2 \right)^2} \right). \tag{A.13}$$

APPENDIX B

WEAK FORMULATION DERIVATION

We derive the general weak formulation for the differential momentum balance (DMB) equation

$$\mathrm{Div}\mathbf{T}_\kappa + \mathbf{b}_\kappa = 0, \ \text{ in } \ \Omega, \tag{B.1}$$

subject to the constitutive equation (Hooke's law)

$$\mathbf{T}_\kappa = 2\mu\mathbf{E} + \lambda \mathrm{tr}(\mathbf{E})\mathbf{I}. \tag{B.2}$$

We first apply Hooke's law to the DMB to obtain

$$\mathrm{Div}(2\mu\mathbf{E} + \lambda\,\mathrm{tr}(\mathbf{E})\mathbf{I}) + \mathbf{b}_\kappa = 0. \tag{B.3}$$

We also note that

$$\mathrm{tr}\,\mathbf{E} = \mathrm{tr}\left(\frac{1}{2}(\nabla\mathbf{u}_\kappa + \nabla\mathbf{u}_\kappa^\mathsf{T})\right) = \frac{1}{2}\,\mathrm{tr}(2\nabla\mathbf{u}_\kappa) = \mathrm{tr}(\nabla\mathbf{u}_\kappa) = \mathrm{Div}\,\mathbf{u}_\kappa. \tag{B.4}$$

This yields

$$\mathrm{Div}(\mu\nabla\mathbf{u}_\kappa) + \mathrm{Div}(\mu\nabla\mathbf{u}_\kappa^\mathsf{T}) + \mathrm{Div}(\lambda\,\mathrm{Div}(\mathbf{u}_\kappa)\mathbf{I}) + \mathbf{b}_\kappa = 0. \tag{B.5}$$

We will make use of several results about how the divergence acts on various products as shown in (Gurtin 2003, p. 30). We list these briefly: *Let* φ, $\mathbf{w}$, *and* $\mathbf{S}$ *be*

a smooth scalar-, vector-, and tensor-valued field, respectively. Then

$$\operatorname{Div}(\varphi\mathbf{w}) = \varphi \operatorname{Div}\mathbf{w} + \mathbf{w}\cdot\nabla\varphi, \tag{B.6}$$

$$\operatorname{Div}(\varphi\mathbf{S}) = \varphi \operatorname{Div}\mathbf{S} + \mathbf{S}\nabla\varphi, \tag{B.7}$$

$$\operatorname{Div}(\mathbf{S}^{\mathsf{T}}\mathbf{w}) = \mathbf{S} : \nabla\mathbf{w} + \mathbf{w}\cdot\operatorname{Div}\mathbf{S}. \tag{B.8}$$

Applying the second of these to the third term in (B.5), and noting that $\operatorname{Div}\mathbf{I} = 0$, we rearrange terms to obtain

$$\nabla(\lambda \operatorname{Div}\mathbf{u}_\kappa) + \operatorname{Div}(\mu\nabla\mathbf{u}_\kappa) + \operatorname{Div}(\mu\nabla\mathbf{u}_\kappa^{\mathsf{T}}) = -\mathbf{b}_\kappa. \tag{B.9}$$

Next, we multiply by a test function $\mathbf{v}$, and integrate over the domain Ω. This yields

$$\int_\Omega \mathbf{v}\cdot\nabla(\lambda \operatorname{Div}\mathbf{u}_\kappa) + \int_\Omega \mathbf{v}\cdot\operatorname{Div}(\mu\nabla\mathbf{u}_\kappa) + \int_\Omega \mathbf{v}\cdot\operatorname{Div}(\mu\nabla\mathbf{u}_\kappa^{\mathsf{T}}) = -\int_\Omega \mathbf{v}\cdot\mathbf{b}_\kappa. \tag{B.10}$$

Applying (B.6) to the first term and (B.8) to the second and third terms yields

$$\begin{aligned}\int_\Omega [\operatorname{Div}((\lambda \operatorname{Div}\mathbf{u}_\kappa)\mathbf{v}) - (\lambda \operatorname{Div}\mathbf{u}_\kappa)\operatorname{Div}\mathbf{v}] + \int_\Omega [\operatorname{Div}(\mu\nabla\mathbf{u}_\kappa^{\mathsf{T}}\mathbf{v}) - \mu\nabla\mathbf{u}_\kappa : \nabla\mathbf{v}] \\ + \int_\Omega [\operatorname{Div}(\mu\nabla\mathbf{u}_\kappa\mathbf{v}) - \mu\nabla\mathbf{u}_\kappa^{\mathsf{T}} : \nabla\mathbf{v}] = -\int_\Omega \mathbf{v}\cdot\mathbf{b}_\kappa.\end{aligned} \tag{B.11}$$

We make use of the Divergence Theorem (see Gurtin 2003, p. 37) three times to obtain

$$\begin{aligned}\int_{\partial\Omega} (\lambda \operatorname{Div}\mathbf{u}_\kappa)\mathbf{v}\cdot\mathbf{n} - \int_\Omega (\lambda \operatorname{Div}\mathbf{u}_\kappa)\operatorname{Div}\mathbf{v} + \int_{\partial\Omega} \mu\nabla\mathbf{u}_\kappa^{\mathsf{T}}\mathbf{v}\cdot\mathbf{n} - \int_\Omega \mu\nabla\mathbf{u}_\kappa : \nabla\mathbf{v} \\ + \int_{\partial\Omega} \mu\nabla\mathbf{u}_\kappa\mathbf{v}\cdot\mathbf{n} - \int_\Omega \mu\nabla\mathbf{u}_\kappa^{\mathsf{T}} : \nabla\mathbf{v} = -\int_\Omega \mathbf{v}\cdot\mathbf{b}_\kappa,\end{aligned} \tag{B.12}$$

where $\mathbf{n}$ is the outward unit normal to $\partial\Omega$. The combined integrand over the boundary is given by

$$\begin{aligned}[\mu(\nabla\mathbf{u}_\kappa + \nabla\mathbf{u}_\kappa^\mathsf{T}) + (\lambda \operatorname{Div}\mathbf{u}_\kappa)\mathbf{I}]\mathbf{v}\cdot\mathbf{n} &= [2\mu\mathbf{E} + \lambda\operatorname{tr}(\mathbf{E})\mathbf{I}]\mathbf{v}\cdot\mathbf{n} \\ &= \mathbf{T}_\kappa\mathbf{v}\cdot\mathbf{n}, \end{aligned} \tag{B.13}$$

using the result in (B.4). Notice that $(\mathbf{S}\mathbf{w})\cdot\mathbf{n} = \mathbf{w}\cdot(\mathbf{S}^\mathsf{T}\mathbf{n})$ for any tensor $\mathbf{S}$ and vectors $\mathbf{w}$ and $\mathbf{n}$. Using this and the fact that $\mathbf{T}_\kappa$ is symmetric, we rearrange terms to obtain

$$\int_\Omega \lambda \operatorname{Div}\mathbf{u}_\kappa \operatorname{Div}\mathbf{v} + \int_\Omega \mu\nabla\mathbf{u}_\kappa : \nabla\mathbf{v} + \int_\Omega \mu\nabla\mathbf{u}_\kappa^\mathsf{T} : \nabla\mathbf{v} - \int_{\partial\Omega} \mathbf{v}\cdot\mathbf{T}_\kappa\mathbf{n} = \int_\Omega \mathbf{v}\cdot\mathbf{b}_\kappa. \tag{B.14}$$

Thus, the general weak formulation is given by

$$a(\mathbf{u}_\kappa, \mathbf{v}) - \int_{\partial\Omega} \mathbf{v}\cdot\mathbf{T}_\kappa\mathbf{n} = (\mathbf{v}, \mathbf{b}_\kappa)_\Omega, \tag{B.15}$$

where $a(\cdot,\cdot)$ is the bilinear form given by

$$a(\mathbf{u}, \mathbf{v}) = (\lambda \operatorname{Div}\mathbf{u}, \operatorname{Div}\mathbf{v})_\Omega + (\mu\nabla\mathbf{u}, \nabla\mathbf{v})_\Omega + (\mu\nabla\mathbf{u}^\mathsf{T}, \nabla\mathbf{v})_\Omega. \tag{B.16}$$

We further note that $a(\cdot,\cdot)$ is typically written in the standard form

$$a(\mathbf{u}, \mathbf{v}) = \int_\Omega \mathbf{S}(\mathbf{u}) : \mathbf{E}(\mathbf{v}) = \int_\Omega \lambda \operatorname{Div}\mathbf{u} \operatorname{Div}\mathbf{v} + \int_\Omega 2\mu\mathbf{E}(\mathbf{u}) : \mathbf{E}(\mathbf{v}), \tag{B.17}$$

where

$$\mathbf{E}(\mathbf{u}) := \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^{\mathsf{T}}), \tag{B.18}$$

$$\mathbf{S}(\mathbf{u}) := 2\mu \mathbf{E}(\mathbf{u}) + \lambda(\mathbf{E}(\mathbf{u}))\mathbf{I}. \tag{B.19}$$

To see this, we apply (B.4) and the facts that $\mathrm{tr}(\mathbf{A}) = \mathrm{tr}(\mathbf{A}^{\mathsf{T}})$ and $\mathrm{tr}(\mathbf{AB}) = \mathrm{tr}(\mathbf{BA})$ to (B.16). Manipulating terms, we obtain

$$\begin{aligned}
a(\mathbf{u}, \mathbf{v}) &= (\lambda \operatorname{Div} \mathbf{u}, \operatorname{Div} \mathbf{v})_\Omega + \frac{1}{2}(\mu \nabla \mathbf{u}, \nabla \mathbf{v})_\Omega + \frac{1}{2}(\mu \nabla \mathbf{u}^{\mathsf{T}}, \nabla \mathbf{v}^{\mathsf{T}})_\Omega \\
&\quad + \frac{1}{2}(\mu \nabla \mathbf{u}^{\mathsf{T}}, \nabla \mathbf{v})_\Omega + \frac{1}{2}(\mu \nabla \mathbf{u}, \nabla \mathbf{v}^{\mathsf{T}})_\Omega \\
&= (\lambda \operatorname{Div} \mathbf{u}, \operatorname{Div} \mathbf{v})_\Omega + 2\mu(\mathbf{E}(\mathbf{u}), \mathbf{E}(\mathbf{v}))_\Omega \\
&= (\lambda \operatorname{Div} \mathbf{u}, \operatorname{tr} \mathbf{E}(\mathbf{v}))_\Omega + 2\mu \int_\Omega \mathbf{E}(\mathbf{u}) : \mathbf{E}(\mathbf{v}) \\
&= \int_\Omega \mathrm{tr}[\lambda(\operatorname{Div} \mathbf{u})\mathbf{E}(\mathbf{v})] + \int_\Omega \mathrm{tr}[2\mu(\mathbf{E}(\mathbf{u}))^{\mathsf{T}}\mathbf{E}(\mathbf{v})] \\
&= \int_\Omega \mathrm{tr}\big([\lambda(\operatorname{tr} \mathbf{E}(\mathbf{u}))\mathbf{I} + 2\mu \mathbf{E}(\mathbf{u})]\mathbf{E}(\mathbf{v})\big) \\
&= \int_\Omega \mathbf{S}(\mathbf{u}) : \mathbf{E}(\mathbf{v}).
\end{aligned} \tag{B.20}$$

APPENDIX C

deal.II BRITTLE FRACTURE CODE

The following listings contain the deal.II (Bangerth, Hartmann, and Kanschat 2007, 2012) files that we created to implement the numerical simulation of brittle fracture as described in Section 3.1. They have been tested on the most recent deal.II release (Version 7.1.0). We briefly summarize these files before presenting the listings:

C.1 ConstST_Fracture.cc The main deal.II program that carries out the numerical simulation. It creates the initial mesh, sets up the finite element and degrees of freedom, assembles the stiffness matrix, and solves the system.

C.2 ConstST_Fracture.prm The parameter handler file that lists the desired run-time parameters. These are read in by the main program.

C.3 create_UCD.h A header file that defines some routines for creating coarsely-meshed hypercubes. We use one of these routines to create the initial grid for our system.

C.4 create_UCD.cc The corresponding source file.

C.5 save_data.h A header file containing some useful tools for saving meshes and their corresponding data vectors.

C.6 save_data.cc The corresponding source file.

C.7 types.h A header file containing the namespace Utils and class DoFVector for combining a DoFHandler, corresponding vector, and corresponding constraint

matrix into a single object to simplify the process of passing these items to various routines.

C.8 `inter_grid_tools.h` A header file containing the namespace `InterGridTools` that provides several functions for synchronizing a volume mesh and a corresponding extracted boundary submesh.

C.9 `inter_grid_tools.cc` The corresponding source file.

C.10 `move_mesh.h` A header file defining the `MoveMesh` class and the namespace `GridUtils` that provide routines for safely moving a mesh by a data vector.

C.11 `move_mesh.cc` The corresponding source file.

C.12 `make_graphs.m` The MATLAB®code used to generate the figures in Section 3.2.

Note that we use ↳ to indicate that the current line is continued from the previous line.

Listing C.1 ConstST_Fracture.cc

```
/** ********************************************************
 * @file   ConstST_Fracture.cc
 * @author Lauren Ferguson
 * @date   Created: March 2012
 * @date   Modified: August 2012
 *
 * @brief
 * This fracture program solves the Griffith crack problem
 * using the Sendova-Walton fracture theory with constant
 * surface tension.
 *
 * @details
 * The two-dimensional classical Griffith crack problem is
 * that of a straight, transverse, quasi-static crack in an
 * infinite linear elastic body subjected to far-field
 * tensile loading.
 *
 * Parameter file: The input parameters for this program must
 * be defined @p ConstST_Fracture.prm

 * Domain:
 * We assume symmetry of the displacement across both the
 * x and y axes to reduce the problem to the upper right
 * quadrant. The problem is solved on the finite computational
 * domain \f$ Q = [0, b]^2 \f$. (Note: we may also choose
 * the domain \f$ Q = [0, b] \times [0, 1] \f$ to reduce
```

```
 * the DOFs since the critical deformation occurs very near
 * to the crack surface.) The right-half of the upper crack
 * surface lies along the x-axis on the interval [0,1], i.e.,
 * the (right) crack tip is located at (1,0).
 * We denote and indicate the pieces of the boundary of Q by
 * - The top face \f$ \Gamma_T \f$, indicated by @p top_id
 * - The left face \f$ \Gamma_L \f$, indicated by @p left_id
 * - The right face \f$ \Gamma_R \f$, not indicated (i.e., 0)
 * - The crack surface \f$ \Gamma_C \f$, indicated by
 *   @p crack_id, and
 * - The bottom surface \f$ \Gamma_B \f$, which is the rest
 *   of the lower face outside the crack surface, indicated
 *   by @p bottom_id
 *
 * Boundary Value Problem:
 * Applying the Sendova-Walton fracture theory to this
 * problem yields the boundary value problem consisting of:
 * 1. A differential momentum balance:
 *      \f[ \mathrm{Div} \mathbf{T}_\kappa
 *          + \mathbf{b}_\kappa = 0 \f]
 *    where \f$ \mathbf{T}_\kappa \f$ is the first
 *    Piola-Kirchhoff stress tensor with components
 *    \f$ \sigma_{ij} \f$ and \f$ \mathbf{b}_\kappa \f$ is a
 *    mutual body force. For now, we assume zero body force.
 *    The subscript \f$ (_\kappa) \f$ indicates that terms
 *    are in the reference configuration.
 * 2. A constitutive equation (Hooke's law):
```

```
 *      \f[ \mathbf{T}_\kappa = 2\mu \mathbf{E}
 *         + \lambda[\mathrm{tr}(\mathbf{E})]\mathbf{I} \f]
 *    where
 *      \f[ \mathbf{E} = 1/2(\nabla \mathbf{u}_\kappa
 *              + \nabla \mathbf{u}_\kappa^T) \f]
 *    is the linearized strain tensor,
 *    \f$ \mathbf{u}_\kappa \f$ is the displacement, and
 *    \f$ \mu, \lambda \f$ are the Lamé material constants.
 * 3. A jump momentum balance in the form of BC on the
 *    crack surface \f$ \Gamma_C \f$:
 *      \f[ \sigma_{12} = 0 \f]
 *      \f[ \sigma_{22} = -\gamma_0 u_{2,11} \f]
 *    where \f$ \gamma_0 \f$ is the constant surface
 *    tension parameter.
 * 4. Symmetry BC on the bottom face \f$ \Gamma_B \f$:
 *      \f[ u_2 = 0 \f]
 *      \f[ \sigma_{12} = 0 \f]
 *    Additional symmetry BC on the left face
 *    \f$ \Gamma_L \f$:
 *      \f[ u_1 = 0 \f]
 *      \f[ u_{2,1} = 0 \f] (equiv: \f[ \sigma_{12} = 0 \f])
 * 5. Far-field loading condition on the top face
 *    \f$ \Gamma_T \f$:
 *      \f[ \sigma_{11} = 0 \f]
 *      \f[ \sigma_{12} = 0 \f]
 *      \f[ \sigma_{22} = \sigma \f]
 *    where \f$ \sigma \f$ is the loading parameter.
```

```
 * 6. Traction-free BC on the right face \f$ \Gamma_R \f$:
 *      \f[ \mathbf{T}_\kappa \mathbf{n} = 0 \f]
 *    where \f$ \mathbf{n} \f$ is the outward unit normal to
 *    the crack surface pointing into the bulk.
 * (Note that all quantities are non-dimensional.)
 *
 * Weak formulation:
 * Together these yield the weak formulation:
 *   \f[
 *   a(\mathbf{u}_\kappa, \mathbf{v})
 *     + \int_{\Gamma_C}\gamma_0 v_{2,1} u_{2,1}
 *   = (\mathbf{v}, \mathbf{b}_\kappa)_Q
 *     + \int_{\Gamma_T} \sigma v_2
 *   \f]
 * where the bilinear form is
 *   \f[
 *   a(\vect{u}, \vect{v}) =
 *   (\lambda\mathrm{Div}\mathbf{u}, \mathrm{Div}\mathbf{v})_Q
 *   + (2\mu\mathbf{E}(\mathbf{u}), \mathbf{E}(\mathbf{v}))_Q.
 *   \f]
 * ********************************************************* */


#include <base/function.h>
#include <base/logstream.h>
#include <base/numbers.h>
#include <base/parameter_handler.h>
#include <base/quadrature_lib.h>
```

```
#include <base/symmetric_tensor.h>
#include <base/timer.h>
#include <base/utilities.h>

#include <lac/constraint_matrix.h>
#include <lac/full_matrix.h>
#include <lac/precondition.h>
#include <lac/solver_cg.h>
#include <lac/solver_gmres.h>
#include <lac/sparse_matrix.h>
#include <lac/vector.h>

#include <grid/grid_generator.h>
#include <grid/grid_in.h>
#include <grid/grid_out.h>
#include <grid/grid_refinement.h>
#include <grid/grid_tools.h>
#include <grid/tria_accessor.h>
#include <grid/tria_boundary_lib.h>
#include <grid/tria.h>
#include <grid/tria_iterator.h>

#include <dofs/dof_accessor.h>
#include <dofs/dof_handler.h>
#include <dofs/dof_tools.h>

#include <fe/fe_q.h>
```

```
#include <fe/fe_system.h>
#include <fe/fe_tools.h>
#include <fe/fe_values.h>

#include <numerics/data_out.h>
#include <numerics/error_estimator.h>
#include <numerics/fe_field_function.h>
#include <numerics/vectors.h>
#include <numerics/matrices.h>

#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>

                  // Additional mesh manipulation tools
#include "types.h"
#include "move_mesh.h"
#include "inter_grid_tools.h"

                  // Mesh creation and data saving tools
#include "create_UCD.h"
#include "save_data.h"

using namespace dealii;
using namespace std;
```

```
/**
 * @namespace ConstST_Fracture
 *
 * @brief
 * Namespace for the @p ConstST_Fracture program, for
 * consistency with @p deal.II practice.
 */
namespace ConstST_Fracture
{



/**
 * This function is identical to that defined in the
 * @p step-18 tutorial program. It defines the (linear)
 * relationship between stress and strain in elasticity, as
 * determined by the Lamé material constants. The resulting
 * rank 4 tensor is of the form
 *   \f[
 *   C_{ijkl} = \mu (\delta_{ik}\delta_{jl}
 *                 + \delta_{il}\delta_{jk})
 *                 + \lambda \delta_{ij} \delta_{kl}
 *   \f]
 * and maps symmetric rank 2 tensors to symmetric rank 2
 * tensors.
 */
```

```
template <int dim>
SymmetricTensor<4,dim>
get_stress_strain_tensor (const double lambda,
                          const double mu)
{
  SymmetricTensor<4,dim> tmp;
  for (unsigned int i=0; i<dim; ++i)
    for (unsigned int j=0; j<dim; ++j)
      for (unsigned int k=0; k<dim; ++k)
        for (unsigned int l=0; l<dim; ++l)
          tmp[i][j][k][l]
            = (((i==k) && (j==l) ? mu : 0.0) +
               ((i==l) && (j==k) ? mu : 0.0) +
               ((i==j) && (k==l) ? lambda : 0.0));

  return tmp;
}



/**
 * As in @p step-18, this function computes the symmetric
 * strain tensor for the given shape function at the given
 * quadrature point using the symmetric gradient of the
 * shape function.
 */
template <int dim>
inline
```

```
SymmetricTensor<2,dim>
get_strain (const FEValues<dim> &fe_values,
            const unsigned int   shape_func,
            const unsigned int   q_point)
{
  SymmetricTensor<2,dim> tmp;

                    // We first fill the diagonal terms which
                    // are simply the derivatives in the ith
                    // direction of the ith component of the
                    // vector-valued shape function. Calling
                    // fe_values.shape_grad_component returns
                    // the full gradient of the ith component
                    // of the shape function at the quadrature
                    // point. (Here, we haven't optimized to
                    // take advantage of the fact that at some
                    // points, the component of the shape
                    // function is always zero.)
  for (unsigned int i=0; i<dim; ++i)
    tmp[i][i] = fe_values.shape_grad_component (shape_func,
    q_point,i)[i];

                    // Then we fill the upper right half of
                    // strain tensor, using symmetry for the
                    // rest.
  for (unsigned int i=0; i<dim; ++i)
    for (unsigned int j=i+1; j<dim; ++j)
```

```
      tmp[i][j] = (fe_values.shape_grad_component (shape_func,
 ↳  q_point,i)[j] +
                   fe_values.shape_grad_component (shape_func,
 ↳  q_point,j)[i]) / 2;

  return tmp;
}



/**
 * As in @p step-18, this function also computes the symmetric
 * strain tensor, but now uses the gradient of a vector-valued
 * field. Given a solution field, calling
 * @p fe_values.get_function_grads extracts gradients of each
 * component of the solution field at a quadrature point,
 * which is returned as a vector of rank 1 tensors (gradient),
 * one each per vector component of the solution. This vector
 * is passed to this function as "grad", and the symmetric
 * strain tensor is constructed by transforming the data
 * storage and symmetrizing.
 */
template <int dim>
inline
SymmetricTensor<2,dim>
get_strain (const vector<Tensor<1,dim> > &grad)
{
  Assert (grad.size() == dim, ExcInternalError());
```

```
  SymmetricTensor<2,dim> strain;
  for (unsigned int i=0; i<dim; ++i)
    strain[i][i] = grad[i][i];

  for (unsigned int i=0; i<dim; ++i)
    for (unsigned int j=i+1; j<dim; ++j)
      strain[i][j] = (grad[i][j] + grad[j][i]) / 2;

  return strain;
}



/**
 * @class ComputePostValues
 *
 * @brief
 * This class is used to compute post-processed quantities
 * that can be locally derived from the solution vector.
 *
 * @details
 * This class (similar to the @p Postprocessor class in
 * @p step-32) is used in conjunction with the
 * @p output_results routine to output the norm of the strain.
 * (You could also output the individual gradient or stress
 * components, which would require a total of dim*dim+1
 * quantities for each quadrature point).
```

```
 *
 * Except for the constructor, the functions here are all
 * called during the call to @p add_data_vector, which is
 * called by the @p DataOut object in @p output_results to add
 * these post-processed values to the output. The actual
 * post-processing work is done in the routine
 * @p compute_derived_quantities_vector
 */
template <int dim>
class ComputePostValues : public DataPostprocessor<dim>
{
  public:

                    /**
                     * Constructor
                     */
    ComputePostValues ();

                    /**
                     * Main function of this class that carries
                     * out the actual computation of post-
                     * processed data. In particular, it
                     * computes the (Frobenius) norm of the
                     * strain at each quadrature point and
                     * stores the results in the
                     * @p computed_quantities vector and is
                     * formatted for output in the
```

```
                 * @p output_results routine.
                 */
  virtual
  void
  compute_derived_quantities_vector(
    const vector<Vector<double> >          &uh,
    const vector<vector<Tensor<1,dim> > > &duh,
    const vector<vector<Tensor<2,dim> > > &dduh,
    const vector<Point<dim> >              &normals,
    const vector<Point<dim> >   &evaluation_points,
    vector<Vector<double> >     &computed_quantities) const;


                /**
                 * Returns a vector of strings representing
                 * the names of the values in
                 * @p computed_quantities.
                 */
  virtual
  vector<string>
  get_names () const;


                /**
                 * Returns the set of @p UpdateFlags that
                 * will be required to compute the desired
                 * quantities, namely @p update_gradients
                 * for our quantities.
                 */
```

```
    virtual
    UpdateFlags
    get_needed_update_flags () const;


                  /**
                   * Returns the number of derived quantities
                   */
    unsigned int
    n_output_variables () const;


                  /**
                   * Returns the format of the quantities to
                   * be output, either scalar or part of a
                   * vector.
                   */
    virtual
    vector<DataComponentInterpretation::
     ↳  DataComponentInterpretation>
    get_data_component_interpretation () const;
};



template <int dim>
ComputePostValues<dim>::ComputePostValues ()
{}
```

```
template <int dim>
void
ComputePostValues<dim>::compute_derived_quantities_vector(
  const vector<Vector<double> >          &/*uh*/,
  const vector<vector<Tensor<1,dim> > > &duh,
  const vector<vector<Tensor<2,dim> > > &/*dduh*/,
  const vector<Point<dim> >              &/*normals*/,
  const vector<Point<dim> >         &/*evaluation_points*/,
  vector<Vector<double> >           &computed_quantities) const
{
  const unsigned int n_quad_points = duh.size();

  Assert(duh.size() == n_quad_points,
         ExcDimensionMismatch (duh.size(), n_quad_points));

  Assert(computed_quantities.size() == n_quad_points,
         ExcDimensionMismatch (computed_quantities.size(),
                               duh.size()));

  Assert(computed_quantities[0].size() == 1,
         ExcInternalError());

  Assert(duh[0].size() == dim,  ExcInternalError());

  Assert (dim ==2, ExcNotImplemented());

  for (unsigned int q=0; q<n_quad_points; ++q)
```

```
  {
                    // Store the (Frobenius) norm of strain:
    computed_quantities[q](0)
      = sqrt (duh[q][0][0]*duh[q][0][0] +
              1/2*pow(duh[q][0][1] + duh[q][1][0], 2) +
              duh[q][1][1]*duh[q][1][1]);
  }
}


template <int dim>
vector<string>
ComputePostValues<dim>::get_names () const
{
  vector<string> names;

  Assert (dim == 2, ExcNotImplemented());

  names.push_back ("norm_of_strain");

  return names;
}



template <int dim>
UpdateFlags
ComputePostValues<dim>::get_needed_update_flags () const
```

```
{
  return update_gradients;
}



template <int dim>
unsigned int
ComputePostValues<dim>::n_output_variables () const
{
  return 1;
}


template<int dim>
vector<DataComponentInterpretation::DataComponentInterpretation
 ↳  >
ComputePostValues<dim>::get_data_component_interpretation ()
 ↳  const
{
  vector<DataComponentInterpretation::
 ↳  DataComponentInterpretation>
        interpretation (1, DataComponentInterpretation::
 ↳  component_is_scalar);

  return interpretation;
}

```

```
/**
 * @class Fracture
 *
 * @brief
 * The main class of the @p ConstST_Fracture program that
 * creates the mesh; sets up, assembles, and solves the
 * system; and outputs results.
 *
 * @details
 * The members of this class are very similar to those of
 * @p step-8 and @p step-18. The significant changes are:
 * - The addition of a specialized mesh generator
 *   @p create_coarse_grid that creates a coarsely-meshed
 *   hypercube and indicates the top, bottom, left, and crack
 *   surfaces on the boundary.
 * - The addition of a parameter handler and corresponding
 *   routine @p declare_parameters to include run-time
 *   parameters.
 * - Two additional output routines:
 *   @p output_deformed_crack that saves the lower edge of
 *   the mesh, including the crack surface, after deformation
 *   by the solution vector; and
 *   @p output_additional_data that creates MATLAB arrays
 *   for further processing of the displacement and slope
 *   u_{2,1} along the crack surface as well as the stress
 *   component sigma_{22} along the bottom face outside the
```

```
 *   crack. It also records the center node and right crack
 *   tip displacement and the crack profile opening angle.
 */
template <int dim>
class Fracture
{
  public:

                    /**
                     * The constructor is similar to that in
                     * @p step-8 where we choose the bilinear
                     * finite element for each of the
                     * @p dim vector components of the
                     * solution.
                     */
    Fracture ();

                    /**
                     * Deconstructor: clears the DoFHandler.
                     */
    ~Fracture ();

                    /**
                     * The driver of this class, called from
                     * the @p main function.
                     */
    void run ();
```

```
private:

                 /**
                  * This function declares the run-time
                  * parameters to be read from the parameter
                  * file @p ConstST_Fracture.prm into the
                  * parameter handler.
                  */
  void declare_parameters ();

                 /**
                   * This function creates the initial
                   * uniformly coarsely meshed hypercube
                   * that is used as the finite
                   * computational domain \f$ Q \f$ for
                   * this problem.
                  */
  void create_coarse_grid ();

                 /**
                  * Refine the grid.
                  */
  void refine_grid ();

                 /**
                  * Set up the data structures for a given
```

```
                 * mesh and distribute the DoFs, similarly
                 * to @p step-8.
                 */
    void setup_system ();

                /**
                 * Assemble the system matrix and RHS
                 * vector, similarly to @p step-18,
                 * using the @p SymmetricTensor class to
                 * assemble the strain tensor.
                 */
    void assemble_system ();

                /**
                 * Solve the global system.
                 */
    void solve_system ();

                /**
                 * Generate graphical output in @p .vtk
                 * format, similarly to @p step-18.
                 */
    void output_results ();

                /**
                 * Output the lower edge of the mesh,
                 * including the crack surface, that has
```

```
                 * been deformed by the displacement
                 * solution.
                 */
void output_deformed_crack ();


                /**
                 * Output additional post-processed data in
                 * table format.
                 */
void output_additional_data (int cycle);


ParameterHandler        prm_handler;

Triangulation<dim>      volume_mesh;
FESystem<dim>           volume_fe;
DoFHandler<dim>         volume_dh;

ConstraintMatrix        hanging_node_constraints;

const QGauss<dim>       quadrature_formula;
const QGauss<dim-1>     face_quadrature_formula;

SparsityPattern         sparsity_pattern;

SparseMatrix<double>    system_matrix;
Vector<double>          volume_rhs;
```

```
  Vector<double>          solution;

                // Global indicator values for the
                // different pieces of the boundary.
                // This allows us to assign numbered ids
                // once and then forget them.
  static const unsigned char crack_id = 1;
  static const unsigned char bottom_id = 2;
  static const unsigned char top_id = 3;
  static const unsigned char left_id = 4;

                // Suffix for output files, depending on
                // the refinement cycle.
  string file_suffix;

                // Base suffix string for MATLAB files and
                // variables, since it doesn't allow
                // periods.
  string matlab_base;

                // Output stream for summary of input and
                // output data.
  ofstream output_stream;

                // Stress-strain tensor, which depends only
                // on the Lamé constants.
  SymmetricTensor<4,dim> stress_strain_tensor;
```

```
};



/**
 * @fn Fracture::Fracture
 *
 * @brief
 * The constructor is similar to that in @p step-8.
 */
template <int dim>
Fracture<dim>::Fracture ()
                :
                volume_fe (FE_Q<dim>(1), dim),
                volume_dh (volume_mesh),
                quadrature_formula (2),
                face_quadrature_formula(2)
{}



/**
 * @fn Fracture::~Fracture
 *
 * @brief
 * Deconstructor: clears the DoFHandler.
 */
template <int dim>
Fracture<dim>::~Fracture ()
```

```
{
  volume_dh.clear ();
}



/**
 * @fn Fracture::declare_parameters
 *
 * @brief
 * This function declares the run-time parameters to
 * be read from the parameter file @p ConstST_Fracture.prm
 * into the parameter handler.
 */
template <int dim>
void Fracture<dim>::declare_parameters ()
{
  // Input Parameters:

  prm_handler.declare_entry("sigma", "0.0",
                            Patterns::Double(0.0, 0.05),
                            "Set tensile loading");

  prm_handler.declare_entry("gamma 0", "0.0",
                            Patterns::Double(0.0, 100.0),
                            "Set surface tension constant");

  prm_handler.declare_entry("lambda", "1",
```

```
                              Patterns::Double(0.0, 1.0),
                              "Set Lamé constant: lambda");

  prm_handler.declare_entry("mu", "1",
                              Patterns::Double(0.0, 1.0),
                              "Set Lamé constant: mu");

  // Mesh and Refinement Parameters:

  prm_handler.declare_entry("body half length", "2",
                              Patterns::Integer(2,100),
                              "Set body half-length");

  prm_handler.declare_entry("body type", "quadrant",
                              Patterns::Selection("quadrant|bar")
↳  ,
                              "Set body type");

  prm_handler.declare_entry("N global refs", "0",
                              Patterns::Integer(0, 10),
                              "Set number of initial global
↳  refinements");

  prm_handler.declare_entry ("N cycles", "2",
                               Patterns::Integer(0,100),
                               "Set number of refinement cycles")
↳  ;
```

```
  prm_handler.declare_entry("refine type", "adaptive",
                            Patterns::Selection("adaptive|
 ↳  global|semi-uniform"),
                            "Set refinement type");

  DataOutInterface<1>::declare_parameters (prm_handler);
}



/**
 * @fn Fracture::create_coarse_grid
 *
 * @brief
 * This function creates the initial uniformly coarsely meshed
 * hypercube that is used as the finite computational domain
 * \f$ Q \f$ for this problem.
 *
 * @details
 * The hypercube is generated from a @p ucd input file created
 * by calling a routine in the @p CreateUCD namespace. The
 * file is then read in using the @p GridIn class.
 *
 * Depending on the desired body type, the generated hypercube
 * is either the full square quadrant \f$ Q = [0,b]^2 \f$,
 * which is an approximation to the upper right quarter-plane
 * (in 2D), or the bar \f$ [0,b] \times [0,1] \f$ which has a
```

```
 * single element in the y direction, where the input
 * parameter \f$ b \f$ is the body half-length.
 */
template <int dim>
void Fracture<dim>::create_coarse_grid ()
{
  string meshfile = "initial_mesh.inp";

                    // The hypercube is \f$ [0,b]^2 \f$
                    // where b is the body_half_length (an
                    // integer > 1). The right half of the
                    // upper crack surface lies on the bottom
                    // face along the interval [0,1], i.e.,
                    // the crack has half-length 1.
  const unsigned int body_half_length =
    prm_handler.get_integer("body half length");

  unsigned int y_length = body_half_length;

  if (prm_handler.get("body type") == "quadrant")
    CreateUCD::uniform_quarter_plane (meshfile, Point<dim>(),
                                      body_half_length);
  else
  {
    CreateUCD::uniform_bar (meshfile, Point<dim>(),
                            body_half_length);
```

```
    y_length = 1;
  }


                  // Read in the mesh and attach it to the
                  // triangulation.
  GridIn<dim> grid_in;
  grid_in.attach_triangulation(volume_mesh);
  ifstream input_stream(meshfile.c_str());
  grid_in.read_ucd(input_stream);
  static const StraightBoundary<dim> boundary;
  volume_mesh.set_boundary (0, boundary);


                  // Next, we identify the different pieces
                  // of the boundary and set their boundary
                  // indicators to the appropriate values.
  typename Triangulation<dim>::active_cell_iterator
    cell = volume_mesh.begin_active(),
    endc = volume_mesh.end();


  for(; cell !=endc; ++cell)
    for(unsigned int face_iter = 0; face_iter < GeometryInfo<
↳   dim>::faces_per_cell; ++face_iter)
      if(cell->face(face_iter)->at_boundary())
      {
                  // Center point of this face
        Point<dim> center = cell->face(face_iter)->center();
```

```
                    // If the face is on bottom boundary, check
                    // if its center point is inside the crack
                    // surface (indicate crack_id) or not
                    // (indicate bottom_id). Else if face is on
                    // top or left boundary, indicate so.
        if(center(dim-1) < .25)
        {
          if ((fabs(center(0)) < 1) && (fabs(center(dim-2)) <
↳  1))
            cell->face(face_iter)->set_boundary_indicator(
↳  crack_id);
          else
            cell->face(face_iter)->set_boundary_indicator(
↳  bottom_id);

        }
        else if(center(dim-1) > y_length - .25)
          cell->face(face_iter)->set_boundary_indicator(top_id)
↳  ;
        else if(center(0) < .25)
          cell->face(face_iter)->set_boundary_indicator(left_id
↳  );
      }

                    // Finally, we refine if desired.
 volume_mesh.refine_global (prm_handler.get_integer("N global
↳  refs"));
```

```
}



/**
 * @fn Fracture::refine_grid
 *
 * @brief
 * Refine the grid.
 *
 * @details
 * The refinement can be adaptive, global, or semi-uniform,
 * depending on the refinement type selected. The adaptive
 * refinement uses the KellyErrorEstimator, similarly to
 * @p step-8, to refine the top 35% percent of cells and
 * coarsen the bottom 35%. The semi-uniform refinement only
 * refines those cells along the crack surface (and any
 * additional cells necessary to keep only one level of
 * hanging nodes) and doesn't coarsen any cells.
 */
template <int dim>
void Fracture<dim>::refine_grid ()
{
  string refine_type = prm_handler.get("refine type");


  if (refine_type == "global")
    volume_mesh.set_all_refine_flags();
  else if (refine_type == "adaptive")
```

```
 {
   Vector<float> estimated_error_per_cell(volume_mesh.
↳  n_active_cells());

   typename FunctionMap<dim>::type neumann_boundary;

   KellyErrorEstimator<dim>::estimate (volume_dh,
                                       face_quadrature_formula,
                                       neumann_boundary,
                                       solution,
                                       estimated_error_per_cell);

   GridRefinement::refine_and_coarsen_fixed_number (
↳  volume_mesh,
                                        estimated_error_per_cell,
                                        0.35, 0.35);
 }
 else if (refine_type == "semi-uniform")
 {
   typename DoFHandler<dim>::active_cell_iterator
     cell = volume_dh.begin_active(),
     endc = volume_dh.end();

   for (; cell != endc; cell++)
     if (cell->at_boundary())
       for (unsigned int face_iter = 0; face_iter <
↳  GeometryInfo<dim>::faces_per_cell; ++face_iter)
```

```
          if(cell->face(face_iter)->boundary_indicator() ==
 ↳  crack_id)
            cell->set_refine_flag();
  }
  else
    AssertThrow(false,
                ExcMessage("Refinement type not recognized"));

  volume_mesh.prepare_coarsening_and_refinement ();
  volume_mesh.execute_coarsening_and_refinement ();
}



/**
 * @fn Fracture::setup_system
 *
 * @brief
 * Set up the data structures for a given mesh
 * and distribute the DoFs, similarly to
 * @p step-8.
 */
template <int dim>
void Fracture<dim>::setup_system ()
{
  volume_dh.clear ();
  volume_dh.distribute_dofs (volume_fe);
```

```
hanging_node_constraints.clear ();
DoFTools::make_hanging_node_constraints (volume_dh,
                                         hanging_node_constraints);
hanging_node_constraints.close ();

sparsity_pattern.reinit(volume_dh.n_dofs(),
                        volume_dh.n_dofs(),
                        volume_dh.max_couplings_between_dofs());

DoFTools::make_sparsity_pattern (volume_dh,
                                 sparsity_pattern);
hanging_node_constraints.condense (sparsity_pattern);

sparsity_pattern.compress();

system_matrix.reinit (sparsity_pattern);
volume_rhs.reinit (volume_dh.n_dofs());
solution.reinit (volume_dh.n_dofs());

                  // Print mesh and DoF info:
cout << "      Number of active cells:       "
     << volume_mesh.n_active_cells() << endl;

cout << "      Number of degrees of freedom: "
     << volume_dh.n_dofs() << endl;

output_stream << "%       Number of active cells:       "
```

```
        << volume_mesh.n_active_cells() << endl;

  output_stream << "%       Number of degrees of freedom: "
        << volume_dh.n_dofs() << endl;
}


/**
 * @fn Fracture::assemble_system
 *
 * @brief
 * Assemble the system matrix and RHS vector,
 * similarly to @p step-18, using the @p SymmetricTensor
 * class to assemble the strain tensor.
 *
 * @details
 * For contributions to the RHS, we will currently
 * use the @p ZeroFunction class, since we assume
 * zero body force. However, this is easily changed
 * for future applications when a nonzero body
 * force is required (for example, see the
 * @p BodyForce class in @p step-18).
 *
 * The main change from @p step-18 is that we must
 * add the contribution of the BC due to the JMB
 * equation on the crack surface. This will
 * involve computing gradients of the shape functions.
```

```
 */
template <int dim>
void Fracture<dim>::assemble_system ()
{
  volume_rhs = 0;
  system_matrix = 0;
  solution = 0;

  FEValues<dim> fe_values (volume_fe,
                           quadrature_formula,
                           update_values |
                           update_gradients |
                           update_quadrature_points |
                           update_JxW_values);

                   // Since we have boundary contributions, we
                   // also need face values.
  FEFaceValues<dim> fe_face_values (volume_fe,
                                    face_quadrature_formula,
                                    update_values |
                                    update_gradients |
                                    update_quadrature_points |
                                    update_JxW_values);

  const unsigned int dofs_per_cell = volume_fe.dofs_per_cell;
  const unsigned int n_q_points = quadrature_formula.size();
```

```
  const unsigned int n_face_q_points = face_quadrature_formula.
 ↳  size();

  FullMatrix<double> cell_matrix (dofs_per_cell,
                                  dofs_per_cell);
  Vector<double>     cell_rhs (dofs_per_cell);

  vector<unsigned int> local_dof_indices (dofs_per_cell);

                   // Get input parameters.
  const double sigma = prm_handler.get_double("sigma");
  const double gamma_0 = prm_handler.get_double("gamma 0");
//   const double mu = prm_handler.get_double("mu");

                   // Assemble the local cell matrix and RHS
                   // vector:
  typename DoFHandler<dim>::active_cell_iterator
    cell = volume_dh.begin_active(),
    endc = volume_dh.end();

  for (; cell!=endc; ++cell)
  {
    cell_matrix = 0;
    cell_rhs = 0;

    fe_values.reinit (cell);
```

```
                    // As in step-18, we use the symmetric
                    // stress-strain tensor to add the
                    // contribution of the bilinear form a(u,v)
                    // to the system matrix.
    for (unsigned int i=0; i<dofs_per_cell; ++i)
      for (unsigned int j=0; j<dofs_per_cell; ++j)
        for (unsigned int q_point=0; q_point<n_q_points; ++
↳   q_point)
          {
            const SymmetricTensor<2,dim>
              eps_phi_i = get_strain (fe_values, i, q_point),
              eps_phi_j = get_strain (fe_values, j, q_point);

            cell_matrix(i,j)
              += (eps_phi_i * stress_strain_tensor * eps_phi_j
                  *
                  fe_values.JxW (q_point));
          }


                    /// We add contributions to both the
                    /// local stiffness matrix and the local
                    /// right-hand side vector due to the
                    /// boundary conditions. Since these only
                    /// occur on the faces, we use
                    /// FEFaceValues to obtain face data.

                    // We also need to keep careful track
```

```
                    // of the components since the
                    // contributions here involve a specific
                    // component of the shape functions, not
                    // the whole vector. Recall for looping
                    // over i and j dofs that i corresponds to
                    // the test function v and j corresponds
                    // to the solution vector u.

    for (unsigned int f=0; f < GeometryInfo<dim>::
↳   faces_per_cell; ++f)
    {
      if(cell->face(f)->at_boundary())
      {
                    /// If the face is on the top boundary, we
                    /// add the contribution from the term
                    /// \f$ \int_{\Gamma_T} \sigma v_2 \f$
                    /// where \f$ \sigma \f$ is the far-field
                    /// tensile loading parameter.
        if(cell->face(f)->boundary_indicator() == top_id)
        {
          fe_face_values.reinit(cell,f);

          for (unsigned int i=0; i<dofs_per_cell; ++i)
          {
            const unsigned int
              component_i = volume_fe.system_to_component_index
↳   (i).first;
```

```
                    // Since we only want v_2, we'll only add
                    // this contribution if component_i == 1.
              if (component_i == 1)
                for (unsigned int q_point=0; q_point <
↳  n_face_q_points; ++q_point)
                  cell_rhs(i) += sigma
                                 * fe_face_values.
↳  shape_value_component(i,q_point,1)
                                 * fe_face_values.JxW(q_point);
            }
        }
                    /// If the face is on the crack boundary,
                    /// we add the contribution of the term
                    /// \f$ \int_{\Gamma_C}\gamma_0 v_{2,1} u_
↳  {2,1} \f$
                    /// where \f$ \gamma_0 \f$ is the constant
                    /// surface tension parameter.
        else if(cell->face(f)->boundary_indicator() == crack_id
↳  )
        {
          fe_face_values.reinit(cell,f);

          for (unsigned int i=0; i<dofs_per_cell; ++i)
            for (unsigned int j=0; j<dofs_per_cell; ++j)
              for (unsigned int q_point=0; q_point<
↳  n_face_q_points; ++q_point)
```

```
            {
              const Tensor<1,dim>
                phi_i_grad = fe_face_values.shape_grad(i,
↳ q_point); // Gradient of v
              const Tensor<1,dim>
                phi_j_grad = fe_face_values.shape_grad(j,
↳ q_point); // Gradient of u

                      // Find the components of the FESystem
                      // corresponding to the dofs
              const unsigned int
                component_i = volume_fe.
↳ system_to_component_index(i).first;
              const unsigned int
                component_j = volume_fe.
↳ system_to_component_index(j).first;

              if ((component_i == 1) && (component_j == 1))
↳ //i.e., v_2 and u_2 components.
                cell_matrix(i,j) += (gamma_0 * phi_i_grad[0]
↳ * phi_j_grad[0]
                                     * fe_face_values.JxW (
↳ q_point));
            }
        }
      }
    }
```

```
                    // Add local contributions to global
                    // counterparts.
    cell->get_dof_indices (local_dof_indices);

    hanging_node_constraints.distribute_local_to_global (
                                          cell_matrix,
                                          local_dof_indices,
                                          system_matrix);

    hanging_node_constraints.distribute_local_to_global (
                                          cell_rhs,
                                          local_dof_indices,
                                          volume_rhs);

  }

                    // Finally, apply the Dirichlet boundary
                    // conditions. First, on the bottom face
                    // outside the crack, we don't want any
                    // movement in the y-direction, so we
                    // fix the dim-1 component of
                    // displacement to be zero.
  map<unsigned int, double> boundary_values;

  vector<bool> component_mask (dim, false);
  component_mask[dim-1] = true;
```

```
 VectorTools::interpolate_boundary_values(volume_dh,
                                          bottom_id,
                                          ZeroFunction<dim>(
↳  dim),
                                          boundary_values,
                                          component_mask);

 MatrixTools::apply_boundary_values (boundary_values,
                                     system_matrix,
                                     solution, volume_rhs);

                 // Similarly, we apply the Dirichlet
                 // condition due to y-symmetry on the left
                 // face.
 component_mask[dim-1] = false;
 component_mask[0] = true;

 VectorTools::interpolate_boundary_values(volume_dh,
                                          left_id,
                                          ZeroFunction<dim>(
↳  dim),
                                          boundary_values,
                                          component_mask);

 MatrixTools::apply_boundary_values (boundary_values,
                                     system_matrix,
```

```
                                        solution, volume_rhs);
}


/**
 * @fn Fracture::solve_system
 *
 * @brief
 * Solve the global system.
 *
 * @details
 * Since the system matrix is symmetric, we solve using the
 * conjugate gradient method, as in @p step-18.
 */
template <int dim>
void Fracture<dim>::solve_system ()
{
  SolverControl             solver_control(volume_dh.n_dofs(),
                                           1e-12);
  SolverCG<>                solver (solver_control);

  PreconditionSSOR<>        preconditioner;
  preconditioner.initialize (system_matrix, 1.2);

  solver.solve (system_matrix, solution, volume_rhs,
                preconditioner);
```

```
  hanging_node_constraints.distribute (solution);

  cout << "   Solver converged in "
       << solver_control.last_step()
       << " iterations" << endl;
  cout << "      L2 norm of Solution: "
       << solution.l2_norm() << endl;

  output_stream << "%    Solver converged in "
                << solver_control.last_step()
                << " iterations with:" << endl;
  output_stream << "%        RHS_L2:       "
                << volume_rhs.l2_norm() << endl;
  output_stream << "%        Solution_L2:  "
                << solution.l2_norm() << endl;
  output_stream << "%        Solution_max: "
                << solution.linfty_norm()<< endl;
}



/**
 * @fn Fracture::output_results
 *
 * @brief
 * Generate graphical output in @p .vtk format,
 * similarly to @p step-18.
 *
```

```
 * @details
 * We output the solution vector on the mesh, i.e., the
 * displacement at each node in each dimensional component,
 * formatted as a vector.
 * We also create a @p ComputePostValues object
 * to output the Frobenius norm of strain as well.
 */
template <int dim>
void
Fracture<dim>::output_results ()
{
  ComputePostValues<dim> post_values;

  DataOut<dim> data_out;
  data_out.attach_dof_handler (volume_dh);

                  // Define names of the solution variables.
  vector<string> solution_names;
  switch (dim)
  {
    case 1:
              solution_names.push_back ("delta_x");
              break;
    case 2:
              solution_names.push_back ("delta_x");
              solution_names.push_back ("delta_y");
              break;
```

```
1200      case 3:
                solution_names.push_back ("delta_x");
                solution_names.push_back ("delta_y");
                solution_names.push_back ("delta_z");
                break;
      default:
                Assert (false, ExcNotImplemented());
    }

    vector<DataComponentInterpretation::
↳   DataComponentInterpretation>
1210        data_component_interpretation(dim,
                DataComponentInterpretation::
↳   component_is_part_of_vector);

    data_out.add_data_vector (solution, solution_names,
                              DataOut<dim>::type_dof_data,
                              data_component_interpretation);

                    // Add the post-processed values to the
                    // DataOut object.
    data_out.add_data_vector (solution, post_values);
1220
    data_out.build_patches ();

    string filename = "ConstST";
    filename += file_suffix;
```

```
  ofstream output (filename.c_str());
  data_out.write_vtk (output);
}



/**
 * @fn Fracture::output_deformed_crack
 *
 * @brief
 * Output the lower edge of the mesh, including the crack
 * surface, that has been deformed by the displacement
 * solution.
 *
 * @details
 * We extract the lower edge of the mesh by creating
 * a new @p FESystem and @p DoFHandler so we can
 * move this submesh by the displacement.
 * Then we find the updated positions of the
 * submesh using a @p FEFieldFunction object.
 */
template <int dim>
void
Fracture<dim>::output_deformed_crack()
{
  Assert(dim ==2, ExcNotImplemented());

                    // First extract the submesh. We want the
```

```
                    // entire lower edge of the mesh, which
                    // corresponds to the crack_id and the
                    // bottom_id. LAF: Not sure why it doesn't
                    // work to use the variable names here.
  set <unsigned char> boundary_ids;
  boundary_ids.insert (1); // crack_id
  boundary_ids.insert (2); // bottom_id

  Triangulation <dim-1, dim> boundary_mesh;
  FESystem<dim-1, dim>   boundary_fe(FE_Q<dim-1,dim>(1), dim);
  DoFHandler<dim-1,dim>  boundary_dh(boundary_mesh);

  map< typename DoFHandler<dim-1, dim>::cell_iterator,
       typename DoFHandler<dim,dim>::face_iterator>
    surface_to_volume_mapping;

  surface_to_volume_mapping
    = GridTools::extract_boundary_mesh (volume_dh,
                                        boundary_dh,
                                        boundary_ids);

                    // Since we may have refined, we need to
                    // synchronize to the current level.
  InterGridTools::synchronize_sub_vertices(volume_dh,
                                   boundary_dh,
                                   surface_to_volume_mapping);
```

```
boundary_dh.distribute_dofs(boundary_fe);

                // Next, we deform the boundary mesh
                // by the displacement.
Functions::FEFieldFunction<dim> u_val (volume_dh, solution);

ConstraintMatrix boundary_constraints;
boundary_constraints.clear ();
DoFTools::make_hanging_node_constraints (boundary_dh,
                                         boundary_constraints);
boundary_constraints.close ();

Vector<double> displacement (boundary_dh.n_dofs());

const unsigned int dofs_per_cell = boundary_fe.dofs_per_cell;
vector<unsigned int> local_dof_indices (dofs_per_cell);

                // We also want to know the number of
                // cells along the crack surface,
                // especially when refining adaptively.
unsigned int n_crack_cells = 0;

typename DoFHandler<dim-1,dim>::active_cell_iterator
  cell=boundary_dh.begin_active(),
  endc=boundary_dh.end();

for (; cell != endc; ++cell)
```

```
  {
                    // If the cell is on the crack surface,
                    // increment the crack cell counter.
    if((cell->face(0)->center())(0) < 1)
      n_crack_cells++;

    cell->get_dof_indices(local_dof_indices);

    for(unsigned int v=0; v<2; ++v)
    {
      Point<dim> v_pt = cell->vertex(v);
      displacement(local_dof_indices[2*v]) = u_val.value(v_pt
↳   ,0);
      displacement(local_dof_indices[2*v+1]) = u_val.value(v_pt
↳   ,1);
    }
  }

  Utils::DoFVector<dim-1,dim> dof_vel(boundary_dh,
                                      displacement,
                                      boundary_constraints);

  GridUtils::move_mesh(dof_vel);

  SaveData::save_grid(boundary_dh, "deformed_crack"+file_suffix
↳   );
```

```
  output_stream << endl
                << "% n_crack_cells: " << setw(5)
                << n_crack_cells << endl;
}



/**
 * @fn Fracture::output_additional_data
 *
 * @brief
 * Output additional post-processed data
 * in table format.
 *
 * @details
 * This routine creates MATLAB arrays for further processing
 * of the displacement and slope u_{2,1} along the crack
 * surface as well as the stress component sigma_{22} along
 * the bottom face outside the crack. It also records the
 * center node and right crack tip displacement and the
 * opening angle (computed from the deformed mesh) of the
 * crack profile.
 */
template <int dim>
void Fracture<dim>::output_additional_data (int cycle)
{
  Functions::FEFieldFunction<dim> u_val (volume_dh, solution);
```

```
                    // First, find the displaced center node
                    // and right crack tip. The
                    // original crack tip is
                    // assumed to be at (1,0).
  Point<dim> right_crack_tip, new_right_crack_tip,
             new_center;

  right_crack_tip[0] = 1;

  new_right_crack_tip[0] = right_crack_tip[0]
                           + u_val.value(right_crack_tip, 0);
  new_right_crack_tip[1] = right_crack_tip[1]
                           + u_val.value(right_crack_tip, 1);
  new_center[0] = u_val.value(Point<dim>(), 0);
  new_center[1] = u_val.value(Point<dim>(), 1);

  output_stream << endl << "% Displaced nodes: " << endl;
  output_stream << "%                    x_val            y_val"
↳  << endl;
  output_stream << "%   center    " << setw(15) << new_center
↳  [0]
                                  << setw(15) << new_center[1] <<
↳   endl;
  output_stream << "%   right tip " << setw(15) <<
↳  new_right_crack_tip[0]
                                  << setw(15) <<
↳  new_right_crack_tip[1] << endl;
```

```
  output_stream << endl;

                  // Next, we find the opening angle of
                  // the crack from the mesh. This is
                  // not particularly accurate, especially
                  // for a coarser mesh.
  Point<dim> R_nearby_point, new_R_nearby_point;

  R_nearby_point[0] =  1 - 1/pow(2,cycle);

  new_R_nearby_point[0] = R_nearby_point[0]
                        + u_val.value(R_nearby_point, 0);
  new_R_nearby_point[1] = R_nearby_point[1]
                        + u_val.value(R_nearby_point, 1);

  double N_R_run = new_right_crack_tip[0]
                 - new_R_nearby_point[0];
  double N_R_rise = new_R_nearby_point[1]
                  - new_right_crack_tip[1];

  double N_R_angle_mesh = atan(N_R_rise/N_R_run)
                        * 180/numbers::PI;

  output_stream << "% Opening angle:" << endl;
  output_stream << "%   right tip" << setw(15) <<
↳  N_R_angle_mesh << endl;
  output_stream << endl;
```

```
                    // Next, we look at u_{2,1} along the
                    // crack surface. We will plot this value
                    // using MATLAB, so we print out these
                    // values for multiple points along
                    // the crack surface and put them in the
                    // form of a MATLAB array.
  unsigned int n_eval_pts = 1000;
  double step_size = 2.0/n_eval_pts;

  Point<dim> eval_pt;
  output_stream << "% Slope: " << endl << endl;
  output_stream << "U = [" << endl;
  for (int i=0; i <= n_eval_pts/2; ++i)
  {
    eval_pt[0] = step_size*i;
    output_stream << step_size*i << ", " << setw(15)
         <<  u_val.gradient(eval_pt,1)[0] << ";" << endl;
  }
  output_stream << "];" << endl << endl;

                    // We also print the necessary statements
                    // to plot the slope and displacement data.
  output_stream << "if (graph_opt == 0)" << endl;
  output_stream << "figure(1);" << endl;
  output_stream << "hold all;" << endl;
  output_stream << "hnew = plot(U(:,1), U(:,2));" << endl;
```

```
  output_stream << "legend('show')" << endl;
  output_stream << "[LEGH,OBJH,OUTH,OUTM] = legend;" << endl;
  output_stream << "legend([OUTH;hnew],OUTM{:},'U" << cycle <<
↳   "')" << endl;
  output_stream << "end" << endl << endl;


                   // Similarly, we plot the crack opening
                   // profile. This is slightly different
                   // because the x-values have changed after
1440               // deformation, as well as the y-values.
                   // Although we are already saving the
                   // crack profile, we still re-save it
                   // here since we can plot it over many
                   // more nodes and get a better picture of
                   // its shape.
  output_stream << "% Profile: " << endl << endl;
  output_stream << "P = [" << endl;
  for (int i=0; i <= n_eval_pts/2; ++i)
  {
1450
    eval_pt[0] = step_size*i;
    output_stream << setw(15) << step_size*i + u_val.value(
↳   eval_pt,0) << ","
                  << setw(15) <<  u_val.value(eval_pt,1)
                  << ";" << endl;
  }
  output_stream << "];" << endl << endl;
```

```
  output_stream << "if (graph_opt == 0)" << endl;
  output_stream << "figure(2);" << endl;
  output_stream << "hold all;" << endl;
  output_stream << "hnew = plot(P(:,1), P(:,2));" << endl;
  output_stream << "legend('show')" << endl;
  output_stream << "[LEGH,OBJH,OUTH,OUTM] = legend;" << endl;
  output_stream << "legend([OUTH;hnew],OUTM{:},'P" << cycle <<
↳   "')" << endl;
  output_stream << "end" << endl << endl;



                   // Finally, we construct similar MATLAB
                   // vectors for the stress component
                   // sigma_{22} outside the crack face.
                   // Recall that stress is given by
                   // T = 2*mu*E + lambda*tr(E)*I.
  double lambda = prm_handler.get_double("lambda");
  double mu = prm_handler.get_double("mu");

  int body_half_length = prm_handler.get_integer("body half
↳   length");
  step_size = (body_half_length - 1.0)/n_eval_pts;

  output_stream << "% Stress: " << endl << endl;

  output_stream << "S = [" << endl;
  for (int i=0; i <= n_eval_pts; ++i)
```

```
  {
    output_stream << 1.0 + step_size*i << "," << setw(15);

    eval_pt[0] = 1.0 + step_size*i;
    output_stream << lambda*u_val.gradient(eval_pt,0)[0]
            + (2*mu+lambda)*u_val.gradient(eval_pt,1)[1]
         << ";" << endl;
  }
  output_stream << "];" << endl << endl;

                  // Again we set up the plots.
  output_stream << "if (graph_opt == 0)" << endl;
  output_stream << "figure(3);" << endl;
  output_stream << "hold all;" << endl;
  output_stream << "hnew = plot(S(:,1), S(:,2));" << endl;
  output_stream << "legend('show')" << endl;
  output_stream << "[LEGH,OBJH,OUTH,OUTM] = legend;" << endl;
  output_stream << "legend([OUTH;hnew],OUTM{:},'S" << cycle <<
↳   "')" << endl;
  output_stream << "end" << endl << endl;

                  // On the last refinement cycle, add labels
                  // and uniquely save the arrays:
  if (cycle == prm_handler.get_integer("N cycles"))
  {
    double sigma = prm_handler.get_double("sigma");
    double gamma_0 = prm_handler.get_double("gamma 0");
```

```
    output_stream << "if (graph_opt == 0)" << endl;
    output_stream << "figure(1);" << endl;
    output_stream << "title('Slope: sigma = " << sigma
                  << ", gamma = " << gamma_0 << "');" << endl;
    output_stream << "xlabel('Position on crack surface')"
                  << endl;
    output_stream << "ylabel('Slope u_{2,1}')"
                  << endl << endl;

    output_stream << "figure(2);" << endl;
    output_stream << "title('Profile: sigma = " << sigma
                  << ", gamma = " << gamma_0 << "');" << endl;
    output_stream << "xlabel('Position on crack surface')"
                  << endl;
    output_stream << "ylabel('Crack profile')"
                  << endl << endl;

    output_stream << "figure(3);" << endl;
    output_stream << "title('Stress: sigma = " << sigma
                  << ", gamma = " << gamma_0 << "');" << endl;
    output_stream << "xlabel('Position on lower edge outside
↳   the crack')" << endl;
    output_stream << "ylabel('Stress sigma_{22}')"
                  << endl;
    output_stream << "end" << endl << endl;
```

```
    output_stream << "U" << matlab_base << " = U(:,2);"
                  << endl;
    output_stream << "P" << matlab_base << " = P(:,2);"
                  << endl;
    output_stream << "S" << matlab_base << " = S(:,2);"
                  << endl;
  }
}



/**
 * @fn Fracture::run
 *
 * @brief
 * The driver of this class, called from the @p main
 * function.
 *
 * @details
 * The run function reads in the input parameters and
 * loops over the refinement cycles, assembling and
 * solving the system during each cycle.
 */
template <int dim>
void Fracture<dim>::run ()
{
  cout << endl << ".....Running ConstST_Fracture Problem ....."
       << endl << endl;
```

```
                    // Load run-time parameters
  declare_parameters ();
  prm_handler.read_input("ConstST_Fracture.prm");

                    // base_suffix = unique identifier
                    // depending on input parameters
  string base_suffix = "_s" + prm_handler.get("sigma")
                       + "_g" + prm_handler.get("gamma 0");

                    // Print a summary of the results to the
                    // results file, in MATLAB format.
                    // MATLAB can't deal with periods in the
                    // filename, so we replace with the letter
                    // 'p' for 'point/period' for this file.
  matlab_base = base_suffix;
  replace(matlab_base.begin(), matlab_base.end(), '.', 'p');
  string matlab_filename = "Results" + matlab_base + ".m";
  output_stream.open(matlab_filename.c_str());
  output_stream << endl
                << "% ***** Results of the ConstST_Fracture
↳  Problem *****"
                << endl << endl;

                    // Define the stress_strain_tensor:
  double lambda = prm_handler.get_double("lambda");
  double mu = prm_handler.get_double("mu");
```

```
 stress_strain_tensor = get_stress_strain_tensor<dim>(lambda,
↳  mu);


                  // Print the run-time parameters to the
1590              // summary output file.
 output_stream << "% INPUT: " << endl << endl;


 output_stream << "% set sigma            = " << prm_handler.
↳  get("sigma") << endl;
 output_stream << "% set gamma 0          = " << prm_handler.
↳  get("gamma 0") << endl;
 output_stream << "% set lambda           = " << lambda <<
↳  endl;
 output_stream << "% set mu               = " << mu << endl;
 output_stream << "% set body half length = " << prm_handler.
↳  get("body half length") << endl;
 output_stream << "% set body type        = " << prm_handler.
↳  get("body type") << endl;
 output_stream << "% set N global refs    = " << prm_handler.
↳  get("N global refs") << endl;
1600  output_stream << "% set N cycles         = " << prm_handler.
↳  get("N cycles") << endl;
 output_stream << "% set refine type      = " << prm_handler.
↳  get("refine type") << endl;


 output_stream << endl << "% OUTPUT:" << endl << endl;
 unsigned int n_cycles = prm_handler.get_integer("N cycles");
```

```
  for (unsigned int cycle=0; cycle <= n_cycles; ++cycle)
  {
    cout << "Refinement cycle: " << cycle << endl;
    output_stream << "% Refinement cycle: " << cycle << endl;

    Timer timer;
    timer.start();

                  // file_suffix = base_suffix + cycle
    file_suffix = base_suffix + "-" + Utilities::int_to_string(
 ↳  cycle,3) + ".vtk";

    if (cycle == 0)
      create_coarse_grid ();
    else
      refine_grid ();

    setup_system ();

                  // Print the initial grid
//     SaveData::save_grid(volume_dh, "initial_mesh"+
 ↳  file_suffix);

    assemble_system ();

    solve_system ();
```

```
1630
    if (cycle == n_cycles) // LAF: only print out last
      output_results ();

    output_deformed_crack ();

    output_additional_data (cycle);

    timer.stop ();
    int time_hr, time_min, time_sec;
1640  time_hr = int(timer.wall_time())/3600;
    time_min = (int(timer.wall_time())%3600)/60;
    time_sec = (int(timer.wall_time())%3600)%60;
    cout << endl << "Timer time: "
         << Utilities::int_to_string(time_hr,2) << ":"
         << Utilities::int_to_string(time_min,2) << ":"
         << Utilities::int_to_string(time_sec,2)<< endl;
    cout << endl << endl;
    output_stream << endl << "% Total Time: "
         << Utilities::int_to_string(time_hr,2) << ":"
1650     << Utilities::int_to_string(time_min,2) << ":"
         << Utilities::int_to_string(time_sec,2)<< endl;
    timer.reset();

    output_stream << endl << endl;
  }
  output_stream.close();
```

```
}



}   // End of ConstST_Fracture namespace



/**
 * @fn main
 * Similarly to many of the tutorial programs, the @p main
 * routine instantiates the @p Fracture class and calls its
 * run function. The dimension is determined from the
 * @p Makefile as @p deal_II_dimension.
 */
int main ()
{
  try
      {
        deallog.depth_console (0);

        const int dim = deal_II_dimension;

        ConstST_Fracture::Fracture<dim>
 ↳  ConstST_Fracture_problem;
        ConstST_Fracture_problem.run ();
      }
  catch (exception &exc)
    {
```

```
      cerr << endl << endl
           << "
↳  ----------------------------------------------------"
           << endl;
      cerr << "Exception on processing: " << endl
           << exc.what() << endl
           << "Aborting!" << endl
           << "
↳  ----------------------------------------------------"
1690       << endl;

      return 1;
    }
  catch (...)
    {
      cerr << endl << endl
           << "
↳  ----------------------------------------------------"
           << endl;
      cerr << "Unknown exception!" << endl
1700       << "Aborting!" << endl
           << "
↳  ----------------------------------------------------"
           << endl;
      return 1;
    }
```

```
  return 0;
}
```

Listing C.2 ConstST_Fracture.prm

```
# -------------------------------------------------------------
# ConstST_Fracture.prm
# Author: Lauren Ferguson
# Date: March 2012
# Modified: August 2012
#
# Parameter handler file for the ConstST_Fracture.cc
# program. Accepted values are in [], default values in ().
# -------------------------------------------------------------

# Input Paramters:
# ----------------

# Far-field (nondimensionalized) tensile loading [(0.0)..0.05]:
  set sigma = 0.05

# Constant surface tension [-100..(0.0)..100]:
  set gamma 0 = 1.0

# Set (nondimensionalized) Lamé constants [0.0..(1.0)]:
# We pick something close to silicon
  set lambda = 0.3501
  set mu = 0.406

# Mesh and Refinement Parameters:
# -------------------
```

```
# Body half-length [(2)..100]
  set body half length = 3

# Body type [(quadrant)|bar]
  set body type = bar

# Number of initial global mesh refinements [(0)..10]:
  set N global refs = 0

# Total number of refinement cycles [0..(2)..100]:
# (one less than the total number of solutions)
  set N cycles = 20

# Refinement type [(adaptive)|global|semi-uniform]
  set refine type = adaptive
```

Listing C.3 `create_UCD.h`

```
/** ***************************************************************
 *
 * @file create_UCD.h
 * @author Lauren Ferguson
 * @date Created: 2011
 * @date Modified: July 2012
 *
 * @brief
 * Contains the namespace @p CreateUCD for functions that
 * create coarsely meshed hypercubes in UCD format.
 *
 * @details
 * The @p CreateUCD functions are used to create specialized
 * hypercubes that can be read in by the @p GridIn routine.
 * The original reason for creating these was that we wanted
 * to use the @p extract_boundary_mesh from the @p GridTools
 * namespace to extract a particular piece of the boundary,
 * namely the crack surface. But since the crack surface
 * doesn't extend across an entire face of the domain, using
 * the regular @p hyper_cube generator from @p GridGenerator
 * doesn't work, since the extraction occurs on the level 0
 * (unrefined) mesh but the crack surface could only be
 * defined after a refinement.
 *
 * So instead of a hypercube that consists of a single cell,
 * these functions create hypercubes that are coarsely meshed,
```

```
 * for which the crack surface, or other desired surface, may
 * be indicated on the level 0 mesh.
 * ********************************************************* */

#ifndef CREATE_UCD_H
#define CREATE_UCD_H

#include <grid/tria.h>
#include <grid/tria_accessor.h>
#include <grid/tria_iterator.h>

#include <iostream>
#include <fstream>

using namespace std;
using namespace dealii;



/**
 * @namespace CreateUCD
 *
 * @brief
 * Namespace for functions that create coarsely meshed
 * hypercubes in UCD format.
 */
namespace CreateUCD
{
```

```
using namespace dealii;
using namespace std;


                 /**
                  * This function creates the hypercube
                  * [-b,b]^(d-1) X [0,b]
                  * where b is @p body_half_length. The
                  * crack center point is mainly given to
                  * pass the value of @p dim, since the
                  * actual crack indication is given in
                  * the calling program. The mesh is
                  * uniform with 2b cells along the
                  * x-axis and b cells along the y-axis.
                  */
template <int dim>
void uniform_half_plane(const string filename,
                        const Point<dim> crack_center,
                        const int body_half_length);


                /**
                  * This function creates the hypercube
                  * [0,b]^d where b is @p body_half_length.
                  * The crack center lies at (0,0) an we
                  * pass it only to pass the value of
                  * @p dim, since the actual crack
                  * indication is given in the calling
```

```
                    * program. The mesh is uniform with
                    * b cells per side.
                    */
  template <int dim>
  void uniform_quarter_plane(const string filename,
                             const Point<dim> crack_center,
                             const int body_half_length);


                    /**
                    * This function creates the hypercube
                    * [0,b] x [0,1] where b is
                    * @p body_half_length. The crack center
                    * lies at (0,0) an we pass it only to pass
                    * the value of @p dim, since the actual
                    * crack indication is given in the calling
                    * program. The mesh is uniform with
                    * b cells along the x-axis and exactly
                    * one cell along the y-axis.
                    */
  template <int dim>
  void uniform_bar(const string filename,
                   const Point<dim> crack_center,
                   const int body_half_length);


}
```

```
#endif
```

Listing C.4 `create_UCD.cc`

```
/** ***************************************************************
 *
 * @file create_UCD.cc
 * @author Lauren Ferguson
 * @date Created: 2011
 * @date Modified: July 2012
 *
 * @brief
 * Contains the namespace @p CreateUCD for functions that
 * create coarsely meshed hypercubes in UCD format.
 *
 * @details
 * We briefly recall the UCD file format:
 *
 * The first section is a single line listing the total number
 * of nodes, the total number of cells, the number of data
 * items per node, and the number of data items per cell.
 *
 * The second sections defines the nodes, with each line
 * listing the node id number and the x-, y-, and
 * z-coordinates.
 *
 * The third section defines the cells, with each line
 * listing the cell id number, the cell material, the cell
 * type, and the associated node ids. We assume all the cells
 * are made of the same material so we set the cell material
```

```
 * to zero for all cells. The cell type we use is @p quad for
 * 2D and @p hex for 3D. The cell itself is defined by the
 * nodes that form its vertices. The order in which we list
 * the nodes must follow the order shown below:
 *
 *         3---2
 *  2D:    |   |
 *         |   |
 *         0---1
 *
 *
 *               7---6
 *              /   /|
 * 3D:         /   / |                                7---6
 *            3---2  5    where the back face has    |   |
 *            |   | /                                |   |
 *            |   |/                                 4---5
 *            0---1
 *
 *
 * Further sections would include node and cell data. We omit
 * these since we do not add any additional data.
 *
 * NOTE: Currently, these functions are only implemented for
 * dim == 2.
 * ********************************************************* */
```

```
#include "create_UCD.h"

using namespace dealii;
using namespace std;



namespace CreateUCD
{

  template <int dim>
  void uniform_half_plane(const string filename,
                          const Point<dim> crack_center,
                          const int body_half_length)
  {
    Assert(crack_center(dim-1) == 0,
           ExcMessage("Crack not on lower edge"));


    cout << ".....Creating UCD file '" << filename
         << "' for uniformly meshed hypercube "
         << "[-" << body_half_length << ","
         << body_half_length << "] X [0,"
         << body_half_length << "]" << endl;

    ofstream ofile(filename.c_str());

    ofile << "# " << filename << endl;
    ofile << "# This file has been generated from the "
```

```
          "CreateUCD::hyper_cube_uniform routine" << endl;
    ofile << "# It creates the uniformly meshed hypercube ";
    ofile << "[-" << body_half_length << ","
          << body_half_length << "] X [0,"
          << body_half_length << "]" << endl;

                  // Number of cells along sides
    const int x_n_cells = 2*body_half_length;
    const int y_n_cells = body_half_length;

                  // Number of nodes/vertices along sides
    const int x_n_nodes = x_n_cells + 1;
    const int y_n_nodes = y_n_cells + 1;

    if (dim == 2)
    {
                  // The first line defines the number of
                  // nodes, cells, and data items
      ofile << x_n_nodes*y_n_nodes <<" "
            << x_n_cells*y_n_cells << " 0 0 0 " << endl;

                  // Next we give the x-, y-, and z-
                  // coordinates for each node.
      for (int i=0; i<y_n_nodes; i++)
        for (int j=0; j<x_n_nodes; j++)
          ofile << i*x_n_nodes + j << " "
                << -body_half_length + j << " "
```

```
                << i << " 0 " << endl;


                  // Next we give the x-, y-, and z-
                  // coordinates for each node.
    for (int i=0; i<y_n_cells; i++)
      for (int j=0; j<x_n_cells; j++)
        ofile << i*x_n_cells + j
              << " 0 quad   "
              << i*x_n_nodes + j << " "
              << i*x_n_nodes + j + 1 << " "
              << (i+1)*x_n_nodes + j + 1 << " "
              << (i+1)*x_n_nodes + j << endl;
  }
  else
    AssertThrow(false, ExcMessage("Invalid dimension in "
                        "CreateUCD::hyper_cube_uniform"));


  ofile.close();
}



template <int dim>
void uniform_quarter_plane(const string filename,
                           const Point<dim> crack_center,
                           const int body_half_length)
{
  Assert(crack_center(dim-1) == 0,
```

```
        ExcMessage("Crack not on lower edge"));

cout << ".....Creating UCD file '" << filename
     << "' for uniformly meshed hypercube "
     << "[0, " << body_half_length << "]^" << dim << endl;

ofstream ofile(filename.c_str());

ofile << "# " << filename << endl;
ofile << "# This file has been generated from the "
         "CreateUCD::hyper_cube_uniform routine" << endl;
ofile << "# It creates the uniformly meshed hypercube ";
ofile << "[0, " << body_half_length << "]^" << dim << endl;

                // Number of cells per side
const int n_cells = body_half_length;

                // Number of nodes/vertices per side
const int n_nodes = n_cells + 1;

if (dim == 2)
{
                // The first line defines the number of
                // nodes, cells, and data items
  ofile << n_nodes*n_nodes <<" "
        << n_cells*n_cells << " 0 0 0 " << endl;
```

```
                    // Next we give the x-, y-, and z-
                    // coordinates for each node.
      for (int i=0; i<n_nodes; i++)
        for (int j=0; j<n_nodes; j++)
          ofile << i*n_nodes + j << " "
                << j << " "
                << i << " 0 " << endl;

                    // Next we give the x-, y-, and z-
                    // coordinates for each node.
      for (int i=0; i<n_cells; i++)
        for (int j=0; j<n_cells; j++)
          ofile << i*n_cells + j
                << " 0 quad   "
                << i*n_nodes + j << " "
                << i*n_nodes + j + 1 << " "
                << (i+1)*n_nodes + j + 1 << " "
                << (i+1)*n_nodes + j << endl;
    }
    else
      AssertThrow(false, ExcMessage("Invalid dimension in "
                                    "CreateUCD::hyper_cube_uniform"));

    ofile.close();
  }
```

```
template <int dim>
void uniform_bar(const string filename,
                 const Point<dim> crack_center,
                 const int body_half_length)
{
  Assert(crack_center(dim-1) == 0,
         ExcMessage("Crack not on lower edge"));

  cout << ".....Creating UCD file '" << filename
       << "' for uniformly meshed hypercube "
       << "[0, " << body_half_length << "] X [0,1]" << endl;

  ofstream ofile(filename.c_str());

  ofile << "# " << filename << endl;
  ofile << "# This file has been generated from the "
           "CreateUCD::hyper_cube_uniform routine" << endl;
  ofile << "# It creates the uniformly meshed hypercube ";
  ofile   << "[0, " << body_half_length << "] X [0,1]" <<
↳  endl;

                 // Number of cells along sides
  const int x_n_cells = body_half_length;
  const int y_n_cells = 1;

                 // Number of nodes/vertices along sides
  const int x_n_nodes = x_n_cells + 1;
```

```
    const int y_n_nodes = y_n_cells + 1;

    if (dim == 2)
    {
                    // The first line defines the number of
                    // nodes, cells, and data items
      ofile << x_n_nodes*y_n_nodes <<" "
            << x_n_cells*y_n_cells << " 0 0 0 " << endl;

                    // Next we give the x-, y-, and z-
                    // coordinates for each node.
      for (int i=0; i<y_n_nodes; i++)
        for (int j=0; j<x_n_nodes; j++)
          ofile << i*x_n_nodes + j << " "
                << j << " "
                << i << " 0 " << endl;

                    // Next we give the x-, y-, and z-
                    // coordinates for each node.
      for (int i=0; i<y_n_cells; i++)
        for (int j=0; j<x_n_cells; j++)
          ofile << i*x_n_cells + j
                << " 0 quad   "
                << i*x_n_nodes + j << " "
                << i*x_n_nodes + j + 1 << " "
                << (i+1)*x_n_nodes + j + 1 << " "
                << (i+1)*x_n_nodes + j << endl;
```

```
    }
    else
      AssertThrow(false, ExcMessage("Invalid dimension in "
                                    "CreateUCD::hyper_cube_uniform"));

    ofile.close();
  }

} // End of CreateUCD namespace


// Explicit Instantiations:

template
void
CreateUCD::uniform_half_plane(const string filename,
                              const Point<2> crack_center,
                              const int body_half_length);

template
void
CreateUCD::uniform_quarter_plane(const string filename,
                                 const Point<2> crack_center,
                                 const int body_half_length);
template
void
CreateUCD::uniform_bar(const string filename,
```

```
                        const Point<2> crack_center,
270                     const int body_half_length);
```

Listing C.5 save_data.h

```
/**
 * @file save_data.h
 * @author Lauren Ferguson
 * @date Created: 2011
 * @date Modified: June 2012
 *
 * @brief
 * Namespace for functions that save meshes and data.
 *
 * @details
 * The functions are used to save the solution or other
 * data given a mesh, a DoFHandler, and a data vector.
 * The output format is @p .vtk unless otherwise stated.
 *
 * @p spacedim = dimension of embedding space (universe)
 * @p dim = dimension of the mesh
 */

#ifndef SAVE_DATA_H
#define SAVE_DATA_H

#include <grid/tria.h>
#include <grid/tria_accessor.h>
#include <grid/tria_iterator.h>
#include <numerics/vectors.h>
```

```
#include <iostream>



/**
 * @namespace SaveData
 *
 * @brief
 * Contains functions for saving meshes and data.
 */
namespace SaveData
{
  using namespace dealii;
  using namespace std;

                   /**
                    * Save the @p data vector to the file
                    * @p filename.
                    */
  template <int dim, int spacedim>
  void save_data (const DoFHandler<dim, spacedim> &dh,
                  Vector<double> &data,
                  const string &filename = "data.vtk",
                  const string &dataname = "data",
                  bool vector_type = 0);

                   /**
                    * Save a zero data vector to the file
```

```
                     * @p filename. This allows one to save
                     * the grid in @p .vtk format.
                     */
  template <int dim, int spacedim>
  void save_grid (const DoFHandler<dim, spacedim> &dh,
                  const string &filename = "grid.vtk");

                    /**
                     * Save the grid in @p .eps format.
                     */
  template <int dim, int spacedim>
  void save_grid_eps (const DoFHandler<dim, spacedim> &dh,
                      const string &filename = "grid.eps");
}



#endif
```

Listing C.6 save_data.cc

```
/**
 * @file save_data.cc
 * @author Lauren Ferguson
 * @date Created: 2011
 * @date Modified: June 2012
 *
 * @brief
 * Contains the namespace @p SaveData for functions that
 * save meshes and data.
 */

#include <grid/grid_out.h>
#include <numerics/data_out.h>

#include <fstream>

#include "save_data.h"

namespace SaveData
{
  using namespace std;
  using namespace dealii;



  template <int dim, int spacedim>
  void save_data (const DoFHandler<dim, spacedim> &dh,
```

```
                    Vector<double> &data,
                    const string &filename,
                    const string &dataname,
                    bool vector_type)
 {

   DataOut<dim, DoFHandler<dim, spacedim> > data_out;

   data_out.attach_dof_handler (dh);

   if (vector_type == 1)
   {
     vector<string> data_names (spacedim, dataname.c_str());
     vector<DataComponentInterpretation::
↳  DataComponentInterpretation>
       data_component_interpretation (spacedim,
         DataComponentInterpretation::
↳  component_is_part_of_vector);

     data_out.add_data_vector (data, data_names,
       DataOut<spacedim,DoFHandler<dim, spacedim> >::
↳  type_dof_data,
       data_component_interpretation);
   }
   else
     data_out.add_data_vector (data, dataname);

```

```
  data_out.build_patches ();

  ofstream output (filename.c_str());
  data_out.write_vtk (output);
}



template <int dim, int spacedim>
void save_grid (const DoFHandler<dim, spacedim> &dh,
                const string &filename)
{
                 // write_vtk expects a data vector,
                 // so we just send zero.
  Vector<double> zero;
  zero.reinit(dh.n_dofs());
  zero = 0;

  DataOut<dim, DoFHandler<dim, spacedim> > data_out;
  data_out.attach_dof_handler (dh);
  data_out.add_data_vector (zero, "zero");
  data_out.build_patches ();

  ofstream output (filename.c_str());
  data_out.write_vtk (output);
}
```

```
  template <int dim, int spacedim>
  void save_grid_eps (const DoFHandler<dim, spacedim> &dh,
                      const string &filename)
  {
    ofstream output (filename.c_str());
    GridOut grid_out;
    grid_out.write_eps (dh.get_tria(), output);
  }


} // End of SaveData namespace


// Explicit Instantiations

template
void SaveData::save_data (const DoFHandler<1, 2> &dh,
                          Vector<double> &data,
                          const string &filename,
                          const string &dataname,
                          bool vector_type);
template
void SaveData::save_data (const DoFHandler<2, 2> &dh,
                          Vector<double> &data,
                          const string &filename,
                          const string &dataname,
```

```
                              bool vector_type);


template
void SaveData::save_grid (const DoFHandler<2, 2> &dh,
                              const string &filename);
template
void SaveData::save_grid (const DoFHandler<1, 2> &dh,
                              const string &filename);


template
void SaveData::save_grid (const DoFHandler<3, 3> &dh,
                              const string &filename);


                      // LAF: So far, I haven't found an easy way
                      // to print out a <1,2> dh mesh...
template
void SaveData::save_grid_eps (const DoFHandler<2, 2> &dh,
                                  const string &filename);
```

Listing C.7 `types.h`

```
/**
 * @file types.h
 * @author Sebastian Pauletti, 2010, 2011
 * @author Modified: Lauren Ferguson, 2012
 *
 * @brief
 * Defines the @p Utils namespace and a class @p DoFVector
 * that combines a @p DoFHandler, corresponding vector, and
 * (optional) corresponding constraint matrix into one object.
 *
 * @details The @p DoFVector should improve legibility and
 * simplify coding, since many routines require these items
 * to be passed.
 *
 * @p spacedim = dimension of embedding space (universe)
 * @p dim = dimension of the mesh
 *
 * LAF: The main modification is the addition of a public bool
 * member, @p hnc_defined, which flags whether or not a
 * constraint matrix has been included. Then routines that
 * receive a @p DoFVector and want to manipulate the matrix
 * can then test for the existence of a constraint matrix.
 *
 * NOTE: Any matrix that is defined for this @p DoFVector MUST
 * be declared previously in the same or a higher scope than
 * the @p DoFVector itself, i.e., you cannot call
```

```
 * DoFVector(my_dof_handler, my_vector, ConstraintMatrix ());
 */

#ifndef TYPES_H
#define TYPES_H

#include <dofs/dof_handler.h>
#include <lac/constraint_matrix.h>
#include <lac/vector.h>



namespace Utils
{
  using namespace dealii;
  using namespace std;

  /**
   * @class DoFVector
   *
   * @brief
   * Combines a DoFHandler, vector, and ConstraintMatrix into
   * a single object for passing to functions that require all
   * three.
   *
   * LAF: Not sure why, but I have problems getting this to
   * work with @p BlockVector.
   */
```

```
 template <int dim, int spacedim=dim, class VECTOR=Vector<
↳  double> >
 class DoFVector
 {
   public:

     typedef VECTOR Vec;

     DoFVector()
     {}

     DoFVector(const DoFHandler<dim,spacedim>  &d,
               VECTOR &v)
        :
        vector(&v),
        dh(&d),
        hanging_node_constraints(NULL),
        hnc_defined(false)
     {
       Assert(vector->size()==dh->n_dofs(),
              ExcInternalError());
     }

     DoFVector(const DoFHandler<dim,spacedim>  &d,
               VECTOR &v,
               const ConstraintMatrix &hnc)
        :
```

```
80          vector(&v),
            dh(&d),
            hanging_node_constraints(&hnc),
            hnc_defined(true)
        {
          Assert(vector->size()==dh->n_dofs(),
                 ExcInternalError());
        }

        void check()
90      {
          Assert(vector->size()==dh->n_dofs(),
                 ExcInternalError());
        }

        void set_vector(VECTOR &v)
        {
          vector = &v;
        }

100     void set_dh(const DoFHandler<dim,spacedim>  &d)
        {
          dh = &d;
        }

        void set_constraints(ConstraintMatrix &hnc)
        {
```

```
    hnc_defined = true;
    hanging_node_constraints = &hnc;
  }

110

  void set_hnc_flag(const bool hnc_flag)
  {
    hnc_defined = hnc_flag;
  }

  const ConstraintMatrix &get_constraints() const
  {
    return (*hanging_node_constraints);
  }

120

  bool get_hnc_flag()
  {
    return (hnc_defined);
  }

  void test ()
  {
    cout << "testing" <<endl;
  }

130

  VECTOR &get_vector()
  {
    return (*vector);
```

```
      }


      VECTOR &get_vector() const
      {
        return (*vector);
      }

      const DoFHandler<dim,spacedim>  &get_dh() const
      {
        return (*dh);
      }

  private:

    VECTOR *vector;
    const DoFHandler<dim,spacedim>  *dh;
    const ConstraintMatrix     *hanging_node_constraints;

  public:

    bool hnc_defined;
  };


} // End of Utils namespace


```

```
#endif
```

Listing C.8 inter_grid_tools.h

```
/**
 * @file inter_grid_tools.h
 * @author Sebastian Pauletti, 2010, 2011
 * @author Modified: Lauren Ferguson, 2012
 *
 * @brief
 * Defines the @p InterGridTools namespace which contains
 * tools to build a dof map between two meshes and
 * synchronize their vertices.
 *
 * @details
 * These tools are used in conjunction with the
 * @p GridTools::extract_boundary_mesh routine. After a mesh
 * extraction, we typically need to synchronize vertices,
 * since the extraction uses the level 0 vertex data only.
 * We use @p synchronize_sub_vertices or
 * @p synchronize_super_vertices to update an extracted
 * submesh or an extractee supermesh, respectively, to the
 * same vertex positions as the corresponding mesh at the
 * current refinement level.
 *
 * The @p build_dof_map creates a mapping between the boundary
 * dof indices and the volume ones.
 *
 * NOTE: None of the synchronization routines correct hanging
 * nodes. It is up to the user to do so after the
```

```
 * synchronization call. This may not be particularly
 * important for synchronizing a submesh, since the supermesh
 * should have already had corrected hanging nodes, but may be
 * more critical for synchronizing a supermesh. The user
 * may want to use the @p correct_hanging_nodes routine
 * from the @p MoveMesh class, which is useful for <2,2>
 * meshes.
 *
 * LAF: MODIFICATIONS:
 *  - The original routines were just @p build_dof_map
 *    and @p synchronize_vertices. We added the "super"
 *    version and so renamed the original as the "sub"
 *    version to emphasize its purpose.
 *  - We also added two new tools so that a volume mesh
 *    and an extracted boundary submesh may be refined
 *    concurrently. The first,
 *    @p set_concurrent_refinement_flags,
 *    ensures that the refinement and coarsening flags of
 *    both meshes agree. The second,
 *    @p update_surface_to_volume_mapping, makes sure that
 *    this mapping includes all the new children that have
 *    been created during refinement and deletes any children
 *    that have been lost due to coarsening.
 *  - The namespace name was changed from @p intergrid
 *  - We also separated the implementation into the
 *    corresponding @p inter_grid_tools.cc file.
 *
```

```
 * NOTE: To call all of these functions, the dh must have
 * already called dh.distribute_dofs(fe).
 *
 * @p spacedim = dimension of embedding space (universe)
 * @p dim = dimension of the mesh
 */

#ifndef INTER_GRID_TOOLS_H
#define INTER_GRID_TOOLS_H

#include <dofs/dof_accessor.h>
#include <dofs/dof_handler.h>
#include <grid/tria.h>


namespace InterGridTools
{
  using namespace dealii;
  using namespace std;

                    /**
                     * Update the vertices of an extracted
                     * submesh with the vertex positions
                     * of the supermesh.
                     */
  template <int dim, int spacedim>
  void
```

```
  synchronize_sub_vertices (
    const DoFHandler<dim, spacedim> &volume_dh,
    const DoFHandler<dim-1, spacedim> &boundary_dh,
    const map< typename DoFHandler<dim-1, spacedim>::
↳   cell_iterator,
    typename DoFHandler<dim, spacedim>::face_iterator >
    &surface_to_volume_mapping);


                  /**
                   * Update the vertices of a supermesh
                   * with the vertex positions of an
                   * extracted submesh.
                   */
  template <int dim, int spacedim>
  void
  synchronize_super_vertices (
    const DoFHandler<dim, spacedim> &volume_dh,
    const DoFHandler<dim-1, spacedim> &boundary_dh,
    const map< typename DoFHandler<dim-1, spacedim>::
↳   cell_iterator,
    typename DoFHandler<dim, spacedim>::face_iterator >
    &surface_to_volume_mapping);


                  /**
                   * Build a map of the dof indices from a
                   * boundary @p DoFHandler to a volume one.
                   */
```

```
  template <int dim, int spacedim>
  map < unsigned int, unsigned int >
  build_dof_map (
    const DoFHandler<dim, spacedim> &volume_dh,
    const DoFHandler<dim-1, spacedim> &boundary_dh,
    const map< typename DoFHandler<dim-1, spacedim>::
↳   cell_iterator,
    typename DoFHandler<dim, spacedim>::face_iterator >
    &surface_to_volume_mapping);


                  /**
                   * Set the refinement and coarsening flags
                   * of an extracted submesh to agree with
                   * those of the supermesh.
                   */
  template <int dim, int spacedim>
  void
  set_concurrent_refinement_flags (
    const DoFHandler<dim, spacedim> &volume_dh,
    DoFHandler<dim-1, spacedim> &boundary_dh,
    map< typename DoFHandler<dim-1, spacedim>::cell_iterator,
    typename DoFHandler<dim, spacedim>::face_iterator >
    &surface_to_volume_mapping);


                  /**
                   * Update the @p surface_to_volume_mapping
                   * after refinement to add in newly
```

```
                    * refined cells and delete coarsened ones.
                    */
  template <int dim, int spacedim>
  void
  update_surface_to_volume_mapping (
    const DoFHandler<dim, spacedim> &volume_dh,
    DoFHandler<dim-1, spacedim> &boundary_dh,
    map< typename DoFHandler<dim-1, spacedim>::cell_iterator,
    typename DoFHandler<dim, spacedim>::face_iterator >
    &surface_to_volume_mapping);
}



#endif
```

Listing C.9 inter_grid_tools.cc

```
/**
 * @file inter_grid_tools.cc
 * @author Sebastian Pauletti, 2010, 2011
 * @author Modified: Lauren Ferguson, 2012
 *
 * @brief
 * Defines the @p InterGridTools namespace which contains
 * tools to build a dof map between two meshes and
 * synchronize their vertices.
 */

#include <dofs/dof_accessor.h>
#include <dofs/dof_handler.h>
#include <grid/tria_accessor.h>
#include <grid/tria_iterator.h>

#include "inter_grid_tools.h"



namespace InterGridTools
{
  using namespace dealii;
  using namespace std;

                    /**
                     * Update the vertices of an extracted
```

```
                     * submesh using the vertex positions of
                     * the supermesh.
                     *
                     * We loop over all the vertices of the
                     * boundary mesh, keeping track of whether
                     * or not they have been updated yet using
                     * the vector of flags @p vertex_touched.
                     * If the vertex hasn't been updated yet,
                     * we update it with the corresponding
                     * volume vertex and set the flag.
                     */
  template <int dim, int spacedim>
  void synchronize_sub_vertices (
    const DoFHandler<dim, spacedim> &/*volume_dh*/,
    const DoFHandler<dim-1, spacedim> &boundary_dh,
    const map< typename DoFHandler<dim-1, spacedim>::
↳   cell_iterator,
    typename DoFHandler<dim, spacedim>::face_iterator >
    &surface_to_volume_mapping)
  {
    vector<bool> vertex_touched (boundary_dh.get_tria().
↳   n_vertices(),
                                 false);

    typename DoFHandler<dim>::face_iterator face;
    typename DoFHandler<dim-1,dim>::active_cell_iterator
      cell = boundary_dh.begin_active(),
```

```
      endc = boundary_dh.end();

    for (; cell != endc; ++cell)
    {
      face = surface_to_volume_mapping.find(cell)->second;

      for (unsigned int v=0; v < GeometryInfo<dim-1>::
↳  vertices_per_cell; ++v)
        if (vertex_touched[cell->vertex_index(v)] == false)
        {
          vertex_touched[cell->vertex_index(v)] = true;
          cell->vertex(v) = face->vertex(v);
        }
    }
  }



                  /**
                   * Update the vertices of the supermesh
                   * using the vertex positions of
                   * the extracted submesh.
                   *
                   * The implementation is the same as in
                   * @p synchronize_sub_vertices except that
                   * the vertex update statement is reversed
                   * to update the volume vertex instead of
                   * the boundary vertex.
```

```
                    */
 template <int dim, int spacedim>
 void synchronize_super_vertices (
   const DoFHandler<dim, spacedim> &/*volume_dh*/,
   const DoFHandler<dim-1, spacedim> &boundary_dh,
   const map< typename DoFHandler<dim-1, spacedim>::
↳  cell_iterator,
   typename DoFHandler<dim, spacedim>::face_iterator >
   &surface_to_volume_mapping)
 {
   vector<bool> vertex_touched (boundary_dh.get_tria().
↳  n_vertices(),
                                false);

   typename DoFHandler<dim>::face_iterator face;
   typename DoFHandler<dim-1,dim>::active_cell_iterator
     cell = boundary_dh.begin_active(),
     endc = boundary_dh.end();

   for (; cell != endc; ++cell)
   {
     face = surface_to_volume_mapping.find(cell)->second;

     for (unsigned int v=0; v < GeometryInfo<dim-1>::
↳  vertices_per_cell; ++v)
       if (vertex_touched[cell->vertex_index(v)] == false)
       {
```

```
          vertex_touched[cell->vertex_index(v)] = true;
          face->vertex(v) = cell->vertex(v);
        }
    }
}



                    /**
                     * To build the dof map, we loop over the
                     * cells of the boundary mesh, find the
                     * corresponding face in the volume mesh
                     * via the @p surface_to_volume_mapping
                     * and then map corresponding dof indices.
                     * The boundary cell dof indices are the
                     * keys in this map, and the volume face
                     * dof indices are the values.
                     */
template <int dim, int spacedim>
map < unsigned int, unsigned int >
build_dof_map (
  const DoFHandler<dim, spacedim> &/*volume_dh*/,
  const DoFHandler<dim-1, spacedim> &boundary_dh,
  const map< typename DoFHandler<dim-1, spacedim>::
↳  cell_iterator,
  typename DoFHandler<dim, spacedim>::face_iterator >
  &surface_to_volume_mapping)
{
```

```
    map < unsigned int, unsigned int > dof_map;

    typename DoFHandler<dim>::face_iterator face;
    typename DoFHandler<dim-1,dim>::active_cell_iterator
      cell = boundary_dh.begin_active(),
      endc = boundary_dh.end();

    for(; cell != endc; ++cell)
    {
      face = surface_to_volume_mapping.find(cell)->second;

      for (unsigned int v=0; v < GeometryInfo<dim-1>::
↳   vertices_per_cell; ++v)
        for (unsigned int d=0; d<dim; ++d)
          dof_map[cell->vertex_dof_index(v,d)] = face->
↳   vertex_dof_index(v,d);
    }

    return dof_map;
  }


                  /**
                   * Set the refinement and coarsening flags
                   * of the boundary mesh to correspond with
                   * those of the volume mesh. This will
                   * enable the user to refine both meshes
```

```
                    * concurrently so that they still match
                    * where they overlap.
                    *
                    * Since we need access to the refinement
                    * flags of volume mesh cells, we'll loop
                    * over volume cells and find the
                    * corresponding faces using the
160                 * @p surface_to_volume_mapping.
                    * Unfortuanately, we can only search for
                    * the keys of this map, not the values,
                    * so we create a second map that switches
                    * these so we can search that mapping for
                    * the correct face. Then we set the
                    * refinement and coarsening flags of the
                    * boundary cell to match those of the
                    * volume.
                    */
170 template <int dim, int spacedim>
void set_concurrent_refinement_flags (
  const DoFHandler<dim, spacedim> &volume_dh,
  DoFHandler<dim-1, spacedim> &/*boundary_dh*/,
  map< typename DoFHandler<dim-1, spacedim>::cell_iterator,
  typename DoFHandler<dim, spacedim>::face_iterator >
  &surface_to_volume_mapping)
{
  map < typename DoFHandler<dim, spacedim>::face_iterator,
```

```
          typename DoFHandler<dim-1, spacedim>::cell_iterator >
↳    reverse_map;
   typename map < typename DoFHandler<dim-1, spacedim>::
↳  cell_iterator,
                  typename DoFHandler<dim, spacedim>::
↳  face_iterator >::iterator it;
   for(it = surface_to_volume_mapping.begin(); it !=
↳  surface_to_volume_mapping.end(); it++)
      reverse_map[it->second]=it->first;

   typename DoFHandler<dim, spacedim>::face_iterator face;
   typename DoFHandler<dim, spacedim>::active_cell_iterator
         vol_cell = volume_dh.begin_active(),
         vol_endc = volume_dh.end();
   typename DoFHandler<dim-1, spacedim>::cell_iterator
↳  bdry_cell;

   for (; vol_cell != vol_endc; ++vol_cell)
     for (unsigned int f=0; f < GeometryInfo<dim>::
↳  faces_per_cell; ++f)
     {
       face = vol_cell->face(f);
       if (reverse_map.find(face) != reverse_map.end())
       {
         bdry_cell = reverse_map.find(face)->second;

         if(vol_cell->refine_flag_set())
```

```
            bdry_cell->set_refine_flag();
          if( vol_cell->coarsen_flag_set() && (!bdry_cell->
↳  refine_flag_set()) )
            bdry_cell->set_coarsen_flag();
        }
      }
 }



                  /**
                   * After the volume and boundary meshes
                   * have been refined, we must update the
                   * @p surface_to_volume_mapping so that it
                   * contains all levels of cell/face
                   * information, including new refinements,
                   * but excludes any cells that were deleted
                   * due to coarsening. To obtain this, we
                   * first create a map with just the level
                   * zero data. Then we swap it with the
                   * @p surface_to_volume_mapping and
                   * add in all the children from all the
                   * refinement levels.
                   */
 template <int dim, int spacedim>
 void update_surface_to_volume_mapping (
   const DoFHandler<dim, spacedim> &/*volume_dh*/,
   DoFHandler<dim-1, spacedim> &boundary_dh,
```

```
    map< typename DoFHandler<dim-1, spacedim>::cell_iterator,
    typename DoFHandler<dim, spacedim>::face_iterator >
    &surface_to_volume_mapping)
  {
    map< typename DoFHandler<dim-1, spacedim>::cell_iterator,
                  typename DoFHandler<dim, spacedim>::
↳   face_iterator > level_zero_map;

    typename DoFHandler<dim-1, spacedim>::cell_iterator cell;

    for (cell = boundary_dh.begin(0); cell != boundary_dh.end
↳   (0); cell++)
      level_zero_map[cell] = surface_to_volume_mapping[cell];

    surface_to_volume_mapping.clear();
    surface_to_volume_mapping.swap(level_zero_map);

    unsigned int level = 0;

    do
    {
      bool add_children = false;

      for (cell = boundary_dh.begin(level); cell != boundary_dh
↳   .end(level); cell++)
        if (cell->has_children() == true)
        {
```

```
250           Assert(surface_to_volume_mapping[cell]->has_children
↳  () == true,
            ExcMessage("Error: Surface and volume meshes "
                       "should have corresponding children."));

          add_children = true;

          for(unsigned int c=0; c < cell->n_children(); c++)
            if(surface_to_volume_mapping.find(cell->child(c))
↳  == surface_to_volume_mapping.end())
              surface_to_volume_mapping[cell->child(c)] =
↳  surface_to_volume_mapping[cell]->child(c);
        }
260

      if(add_children == true)
        level++;
      else
        break;

    } while (true);
  }



270 } // End of InterGridTools namespace


// Explicit Instantiations:
```

```
template
void
InterGridTools::synchronize_sub_vertices (
  const DoFHandler<2, 2> &volume_dh,
  const DoFHandler<1, 2> &boundary_dh,
  const map< typename DoFHandler<1, 2>::cell_iterator,
  typename DoFHandler<2, 2>::face_iterator >
  &surface_to_volume_mapping);

template
void
InterGridTools::synchronize_super_vertices (
  const DoFHandler<2, 2> &volume_dh,
  const DoFHandler<1, 2> &boundary_dh,
  const map< typename DoFHandler<1, 2>::cell_iterator,
  typename DoFHandler<2, 2>::face_iterator >
  &surface_to_volume_mapping);

template
std::map < unsigned int, unsigned int >
InterGridTools::build_dof_map (
  const DoFHandler<2, 2> &volume_dh,
  const DoFHandler<1, 2> &boundary_dh,
  const map< typename DoFHandler<1, 2>::cell_iterator,
  typename DoFHandler<2, 2>::face_iterator >
  &surface_to_volume_mapping);
```

```
template
void
InterGridTools::set_concurrent_refinement_flags (
  const DoFHandler<2, 2> &volume_dh,
  DoFHandler<1, 2> &boundary_dh,
  map< typename DoFHandler<1, 2>::cell_iterator,
  typename DoFHandler<2, 2>::face_iterator >
  &surface_to_volume_mapping);

template
void
InterGridTools::update_surface_to_volume_mapping (
  const DoFHandler<2, 2> &volume_dh,
  DoFHandler<1, 2> &boundary_dh,
  map< typename DoFHandler<1, 2>::cell_iterator,
  typename DoFHandler<2, 2>::face_iterator >
  &surface_to_volume_mapping);
```

Listing C.10 move_mesh.h

```
/**
 * @file move_mesh.h
 * @author Sebastian Pauletti, 2010, 2011
 * @author Edited: Lauren Ferguson, 2012
 *
 * @brief
 * Defines the @p MoveMesh class and @p GridUtils namepsace
 * to provide safe and user-transparent motions of the mesh.
 */

#ifndef MOVE_MESH_H
#define MOVE_MESH_H

#include <fe/fe_system.h>
#include <dofs/dof_handler.h>
#include <grid/tria.h>
#include <lac/constraint_matrix.h>
#include <lac/vector.h>

#include "types.h"

using namespace dealii;
using namespace std;



/**
```

```
 * @class MoveMesh
 *
 * @brief
 * Provides safe and user-transparent
 * motions of the mesh by properly taking care of hanging
 * nodes and interpolations and projections between vectors
 * and vertex coordinates.
 *
 * @details
 * For the general case in which the computational domain is
 * given by a map of the reference cell
 *   \f$ \Omega = F(\hat\Omega) \f$,
 * and the increment function
 *   \f$ V:\Omega\to spacedim \f$,
 * then the mesh motion is given by the map update
 *   \f$ F = F + V(F) \f$.
 * In general, this is not in the space of the map.
 *
 * We currently restrict this class to the case when the map
 * is Q1. For this case, the motion of the mesh can be
 * attained by moving the vertices in the following way:
 *   <code> vertex[i] = vertex[i] + V[table[i]] </code>
 * where @p table contains the mapping from vertex indices
 * to Q1 dofs.
 *
 * The functions for moving the mesh follow the general
 * pattern of operations:
```

```
 *    <ul>
 *    <li> get the coordinates
 *    <li> operate on the vector
 *    <li> write back to the vertices
 *    </ul>
 */
template <int dim, int spacedim>
class MoveMesh
{
  public:

                    /** Constructor */
    MoveMesh (Triangulation<dim,spacedim> &tria);

                    /** @name Moving the mesh */
                    /*@{*/

                    /**
                     * Move the mesh a displacement equal to
                     *   <i> velocity * tau
                     * where velocity is a vector and tau a
                     * time increment.
                     */
    void move(const Utils::DoFVector<dim,spacedim, Vector<
↳   double> > &dof_vel,
              const double tau = 1.0);
```

```
80                  /**
                     * Update the mesh, keeping the
                     * same connectivity but using the
                     * new coordinates provided in X.
                     */
    void update(const Utils::DoFVector<dim,spacedim, Vector<
↳   double> > &dof_x);

                    /**
                     * Update the mesh, keeping the
                     * same connectivity but using the
90                   * new coordinates provided in X.
                     */
    void normal_update(const Utils::DoFVector<dim,spacedim,
↳   Vector<double> > &dof_x,
                       const Utils::DoFVector<dim,spacedim,
↳   Vector<double> > &dof_n);
                    /*@}*/

                    /** Reset if the mesh is adapted */
    void reset();

  protected:
100
                    /** Internal function to fill a Vector
                     * with the coordinates of the vertices */
```

```
    void get_coords(const Utils::DoFVector<dim,spacedim, Vector
↳   <double> > &dof_x);

                    /** Internal function to set the
                     * coordinates of the vertices with
                     * Vector */
    void set_coords(const Utils::DoFVector<dim,spacedim,
                     Vector<double> > &dof_x);
110

  public:

    void correct_hanging_nodes();

                    /** @name Implementation and testing */
                    /*@{*/

                    /**
                     * Save the vertices coordinates to a
120                  * file.
                     */
    void save_coordinates (const string &fname);

                    /**
                     * Set the vertices coordinates from the
                     * Vector X and save them to a file.
                     */
    void save_coordinates (const string &fname,
```

```
                              const Utils::DoFVector<dim,spacedim,
                              Vector<double> > &X);
                    /*@}*/

  protected:

                    // Coordinates
    FESystem <dim,spacedim>   fe_coord;
    DoFHandler<dim,spacedim>  dh_coord;
    Vector<double>            coord;

    vector< vector<unsigned int> > index_to_dof;

    ConstraintMatrix     hanging_node_constraints;
};



/**
 * @namespace GridUtils
 *
 * @brief
 * A collection of useful function templates to move meshes.
 */
namespace GridUtils
{
  using namespace dealii;
  using namespace std;
```

```
                    /**
                     * Move the mesh for a displacement
                     * vel * tau, where vel is a velocity and
                     * tau a time increment.
                     */
  template <int dim, int spacedim>
  void move_mesh(const Utils::DoFVector<dim,spacedim, Vector<
↳   double> > &vel,
                 const double tau = 1.0);


                    /**
                     * Update the mesh, keeping the same
                     * connectivity but using the new
                     * coordinates provided in X.
                     */
  template <int dim, int spacedim>
  void update_mesh(const Utils::DoFVector<dim,spacedim, Vector<
↳   double> > &X);


                    /**
                     * Fill the Vector X with the coordinates
                     * of the mesh.
                     */
  template <int dim, int spacedim>
  void interpolate_coordinates(Utils::DoFVector<dim,spacedim,
↳   Vector<double> > &X);
```

```
180

  template <int dim, int spacedim>
  void normal_update(const Utils::DoFVector<dim,spacedim,
 ↳  Vector<double> > &normal1,
                     const Utils::DoFVector<dim,spacedim,
 ↳  Vector<double> > &X1);
}



#endif
```

Listing C.11 `move_mesh.cc`

```
/**
 * @file move_mesh.cc
 * @author Sebastian Pauletti, 2010, 2011
 * @author Modified: Lauren Ferguson, 2012
 *
 * @brief
 * Defines the @p MoveMesh class and @p GridUtils namepsace
 * to provide safe and user-transparent motions of the mesh.
 *
 * @details
 * LAF: The main modification is the use of the flag
 * @p hnc_defined to ensure that functions requiring hanging
 * node constraints have them available. In particular,
 * @p FETools::interpolate gives an error if you send
 * an non-initialized constraint matrix.
 */

#include <base/quadrature_lib.h>
#include <grid/tria_iterator.h>
#include <dofs/dof_accessor.h>
#include <dofs/dof_tools.h>
#include <fe/fe_q.h>
#include <fe/fe_tools.h>
#include <lac/vector.h>
#include <numerics/data_out.h>
```

```
#include <iostream>
#include <fstream>
#include <vector>

#include "move_mesh.h"

using namespace dealii;
using namespace std;



template <int dim, int spacedim>
MoveMesh<dim, spacedim>::MoveMesh(Triangulation<dim,spacedim> &
 ↳  tria)
  :
  fe_coord (FE_Q<dim,spacedim>(1), spacedim),
  dh_coord (tria)
{
  reset();
}



template <int dim, int spacedim>
void MoveMesh<dim, spacedim>::reset()
{
  dh_coord.distribute_dofs (fe_coord);
  coord.reinit (dh_coord.n_dofs());
```

```
  hanging_node_constraints.clear ();
  DoFTools::make_hanging_node_constraints (dh_coord,
 ↳  hanging_node_constraints);
  hanging_node_constraints.close ();
}



template <int dim, int spacedim>
void MoveMesh<dim, spacedim>::get_coords(
  const Utils::DoFVector<dim,spacedim, Vector<double> > &dof_x)
{
  vector<bool> vertex_touched (dh_coord.get_tria().n_vertices()
 ↳  ,false);

  for (typename DoFHandler<dim,spacedim>::active_cell_iterator
       cell = dh_coord.begin_active ();
       cell != dh_coord.end(); ++cell)
    for (unsigned int v=0; v<GeometryInfo<dim>::
 ↳  vertices_per_cell; ++v)
      if (vertex_touched[cell->vertex_index(v)] == false)
      {
        vertex_touched[cell->vertex_index(v)] = true;
        for (unsigned int d=0; d<spacedim; ++d)
          coord(cell->vertex_dof_index(v,d)) = cell->vertex(v)[
 ↳  d];
      }
```

```
  const DoFHandler<dim,spacedim> &dh_x   = dof_x.get_dh();
  Vector<double>                 &x      = dof_x.get_vector();
  const ConstraintMatrix         &hnc_x = dof_x.get_constraints
 ↳  ();

                  // LAF: we must check whether hnc_x is
                  // empty or not before calling this
                  // interpolate function.
  if (dof_x.hnc_defined)
    FETools::interpolate(dh_coord, coord, dh_x, hnc_x, x);
  else
    FETools::interpolate(dh_coord, coord, dh_x, x);
}



template <int dim, int spacedim>
void MoveMesh<dim, spacedim>::set_coords(
  const Utils::DoFVector<dim,spacedim, Vector<double> > &dof_x)
{
  const DoFHandler<dim,spacedim> &dh    = dof_x.get_dh();
  const Vector<double>           &x     = dof_x.get_vector();
  FETools::interpolate(dh, x, dh_coord,
 ↳  hanging_node_constraints, coord);

  vector<bool> vertex_touched (dh_coord.get_tria().n_vertices()
 ↳  ,false);
```

```
  for (typename DoFHandler<dim,spacedim>::active_cell_iterator
       cell = dh_coord.begin_active ();
       cell != dh_coord.end(); ++cell)
    for (unsigned int v=0; v<GeometryInfo<dim>::
↳  vertices_per_cell; ++v)
      if (vertex_touched[cell->vertex_index(v)] == false)
      {
        vertex_touched[cell->vertex_index(v)] = true;
        for (unsigned int d=0; d<spacedim; ++d)
          cell->vertex(v)[d] = coord(cell->vertex_dof_index(v,d
↳  ));
      }
}



template <int dim, int spacedim>
void MoveMesh<dim, spacedim>::correct_hanging_nodes ()
{
  Vector <double> x(coord.size());
  Utils::DoFVector<dim,spacedim,Vector<double> >
    dof_x(dh_coord,x,hanging_node_constraints);
  get_coords(dof_x);
  set_coords(dof_x);
}



template <int dim, int spacedim>
```

```
void MoveMesh<dim, spacedim>::save_coordinates (const string &
 ↳  fname)
{

  Vector <double> x(coord.size());
  Utils::DoFVector<dim,spacedim,Vector<double> >
    dof_x(dh_coord,x,hanging_node_constraints);
  get_coords(dof_x);

  vector<DataComponentInterpretation::
 ↳  DataComponentInterpretation>
    data_component_interpretation
    (spacedim, DataComponentInterpretation::
 ↳  component_is_part_of_vector);

  vector<string> solution_names (spacedim, "Coordinates");

  DataOut<dim, DoFHandler<dim,spacedim> > data_out;
  data_out.attach_dof_handler (dh_coord);
  data_out.add_data_vector (coord, solution_names,
                            DataOut<dim,DoFHandler<dim,
 ↳  spacedim> >::type_dof_data,
                            data_component_interpretation);
  data_out.build_patches ();

  ofstream output (fname.c_str());
  data_out.write_vtk (output);
```

```
}


150

template <int dim, int spacedim>
void MoveMesh<dim, spacedim>::save_coordinates (
  const string &fname,
  const Utils::DoFVector<dim,spacedim, Vector<double> > &X)
{
  set_coords(X);
  save_coordinates (fname);
}


160

template <int dim, int spacedim>
void MoveMesh<dim, spacedim>::move(
  const Utils::DoFVector<dim,spacedim, Vector<double> > &
 ↳  dof_vel,
  const double tau)
{
  const DoFHandler<dim,spacedim> &dh_vel  = dof_vel.get_dh();
  const Vector<double>           &vel     = dof_vel.get_vector
 ↳  ();
  const ConstraintMatrix         &hnc_vel = dof_vel.
 ↳  get_constraints();


170  Vector <double> x(vel.size());
```

```
  Utils::DoFVector<dim,spacedim, Vector<double> > dof_x(dh_vel,
↳  x,hnc_vel);

                    // LAF: If get_constraints returns null,
                    // we'll have a problem when we call
                    // get_coords. So we need to update the
                    // flag indicating whether or not the
                    // constraints are defined.
  dof_x.set_hnc_flag(dof_vel.hnc_defined);

                    // LAF: we can't call set_/get_hnc_flag of
                    // dof_vel here for some reason, so we
                    // directly access the public member
                    // hnc_defined.
  get_coords(dof_x);

  Vector <double> tmp_vel(vel);
  tmp_vel *= tau;
  x += tmp_vel;

  set_coords(dof_x);
}



template <int dim, int spacedim>
void MoveMesh<dim, spacedim>::update(
  const Utils::DoFVector<dim,spacedim, Vector<double> > &dof_x)
```

```
{
  set_coords(dof_x);
}


template <int dim, int spacedim>
void MoveMesh<dim, spacedim>::normal_update(
  const Utils::DoFVector<dim,spacedim, Vector<double> > &dof_x,
  const Utils::DoFVector<dim,spacedim, Vector<double> > &dof_n)
{

  const DoFHandler<dim,spacedim> &dh_x  = dof_x.get_dh();
  const Vector<double>           &x     = dof_x.get_vector();
  const Vector<double>           &n     = dof_n.get_vector();

  vector<bool> vertex_touched (dh_x.get_tria().n_vertices(),
 ↳  false);

  for (typename DoFHandler<dim,spacedim>::active_cell_iterator
       cell = dh_x.begin_active ();
       cell != dh_x.end(); ++cell)
    for (unsigned int v=0; v<GeometryInfo<dim>::
 ↳  vertices_per_cell; ++v)
      if (vertex_touched[cell->vertex_index(v)] == false)
      {
        vertex_touched[cell->vertex_index(v)] = true;
```

```
        Point<spacedim> vertex_inc;
        for (unsigned int d=0; d<spacedim; ++d)
          vertex_inc[d] = (x(cell->vertex_dof_index(v,d))-cell
 ↳  ->vertex(v)[d]);

        double a = 0.;
        for (unsigned int d=0; d<spacedim; ++d)
          a += vertex_inc[d]*n(cell->vertex_dof_index(v,d));

230     for (unsigned int d=0; d<spacedim; ++d)
          vertex_inc[d] = a*n(cell->vertex_dof_index(v,d));

        cell->vertex(v) += vertex_inc;
      }
}



template <int dim, int spacedim>
void GridUtils::move_mesh(
240   const Utils::DoFVector<dim,spacedim, Vector<double> > &vel1,
  const double tau)
{
  const DoFHandler<dim,spacedim>& dof_handler = vel1.get_dh();
  const Vector<double> &vel = vel1.get_vector();

  vector<bool> vertex_touched (dof_handler.get_tria().
 ↳  n_vertices(), false);
```

```
  for (typename DoFHandler<dim,spacedim>::active_cell_iterator
       cell = dof_handler.begin_active ();
       cell != dof_handler.end(); ++cell)
    for (unsigned int v=0; v<GeometryInfo<dim>::
↳  vertices_per_cell; ++v)
      if (vertex_touched[cell->vertex_index(v)] == false)
      {
        vertex_touched[cell->vertex_index(v)] = true;

        Point<spacedim> vertex_displacement;
        for (unsigned int d=0; d<spacedim; ++d)
          vertex_displacement[d] = tau * vel(cell->
↳  vertex_dof_index(v,d));

        cell->vertex(v) += vertex_displacement;
      }
}



template <int dim, int spacedim>
void GridUtils::update_mesh(
  const Utils::DoFVector<dim,spacedim, Vector<double> > &X1)
{
  const DoFHandler<dim,spacedim>& dof_handler = X1.get_dh();
  const Vector<double> &X = X1.get_vector();
```

```
  vector<bool> vertex_touched (dof_handler.get_tria().
 ↳  n_vertices(), false);

  for (typename DoFHandler<dim,spacedim>::active_cell_iterator
       cell = dof_handler.begin_active ();
       cell != dof_handler.end(); ++cell)
    for (unsigned int v=0; v<GeometryInfo<dim>::
 ↳  vertices_per_cell; ++v)
      if (vertex_touched[cell->vertex_index(v)] == false)
      {
        vertex_touched[cell->vertex_index(v)] = true;

        Point<spacedim> vertex_coord;
        for (unsigned int d=0; d<spacedim; ++d)
          vertex_coord[d] = X(cell->vertex_dof_index(v,d));

        cell->vertex(v) = vertex_coord;
      }
}


template <int dim, int spacedim>
void GridUtils::normal_update(
  const Utils::DoFVector<dim,spacedim, Vector<double> > &
 ↳  normal1,
  const Utils::DoFVector<dim,spacedim, Vector<double> > &X1)
{
```

```
  // TODO: normal should be interpolated to the right space

  const DoFHandler<dim,spacedim>& dof_handler = X1.get_dh();
  const Vector<double> &X = X1.get_vector();
  const Vector<double> &normal = normal1.get_vector();

  vector<bool> vertex_touched (dof_handler.get_tria().
↳  n_vertices(), false);

  for (typename DoFHandler<dim,spacedim>::active_cell_iterator
       cell = dof_handler.begin_active ();
       cell != dof_handler.end(); ++cell)
    for (unsigned int v=0; v<GeometryInfo<dim>::
↳  vertices_per_cell; ++v)
      if (vertex_touched[cell->vertex_index(v)] == false)
      {
        vertex_touched[cell->vertex_index(v)] = true;

        Point<spacedim> vertex_inc;// = cell->vertex(v);
        for (unsigned int d=0; d<spacedim; ++d)
          vertex_inc[d]
          = (X(cell->vertex_dof_index(v,d))-cell->vertex(v)[d])
↳  ;

        double a = 0.;
        for (unsigned int d=0; d<spacedim; ++d)
```

```
          a += vertex_inc[d]*normal(cell->vertex_dof_index(v,d)
 ↳  );

        for (unsigned int d=0; d<spacedim; ++d)
          vertex_inc[d] = a*normal(cell->vertex_dof_index(v,d))
 ↳  ;

        cell->vertex(v) += vertex_inc;
      }
}



template <int dim, int spacedim>
void GridUtils::interpolate_coordinates(
  Utils::DoFVector<dim,spacedim, Vector<double> > &X1)
{
  //TODO: Add some tests

  const DoFHandler<dim,spacedim>& dof_handler = X1.get_dh();
  Vector<double> &X = X1.get_vector();

  QTrapez<dim> quadrature_formula;

  vector<bool> vertex_touched (dof_handler.get_tria().
 ↳  n_vertices(), false);

  for (typename DoFHandler<dim,spacedim>::active_cell_iterator
```

```
       cell = dof_handler.begin_active ();
       cell != dof_handler.end(); ++cell)
    for (unsigned int v=0; v<GeometryInfo<dim>::
↳   vertices_per_cell; ++v)
    {
      const unsigned int vertex_index = cell->vertex_index(v);

      if (vertex_touched[vertex_index] == false)
      {
        vertex_touched[vertex_index] = true;
        Point<spacedim> vertex_coord = cell->vertex(v);
        for (unsigned int d=0; d<spacedim; ++d)
          X(cell->vertex_dof_index(v,d)) = vertex_coord[d];
      }
    }
}



// Explicit Instantiations:

template void
GridUtils::move_mesh(
  const Utils::DoFVector<1,2, Vector<double> > &vel,
  const double tau);

template void
GridUtils::move_mesh(
```

```
  const Utils::DoFVector<2,2, Vector<double> > &vel,
  const double tau);

template void
GridUtils::move_mesh(
  const Utils::DoFVector<2,3, Vector<double> > &vel,
  const double tau);

template void
GridUtils::move_mesh(
  const Utils::DoFVector<3,3, Vector<double> > &vel,
  const double tau);



template void
GridUtils::update_mesh(
  const Utils::DoFVector<1,2, Vector<double> > &X);

template void
GridUtils::update_mesh(
  const Utils::DoFVector<2,2, Vector<double> > &X);

template void
GridUtils::update_mesh(
  const Utils::DoFVector<2,3, Vector<double> > &X);

template void
```

```
GridUtils::update_mesh(
  const Utils::DoFVector<3,3, Vector<double> > &X);



template void
GridUtils::interpolate_coordinates(
  Utils::DoFVector<1,2, Vector<double> > &X);

template void
GridUtils::interpolate_coordinates(
  Utils::DoFVector<2,2, Vector<double> > &X);

template void
GridUtils::interpolate_coordinates(
  Utils::DoFVector<2,3, Vector<double> > &X);

template void
GridUtils::interpolate_coordinates(
  Utils::DoFVector<3,3, Vector<double> > &X);



template void
GridUtils::normal_update(
  const Utils::DoFVector<1,2, Vector<double> > &N,
  const Utils::DoFVector<1,2, Vector<double> > &X);

template void
```

```
GridUtils::normal_update(
  const Utils::DoFVector<2,3, Vector<double> > &N,
  const Utils::DoFVector<2,3, Vector<double> > &X);


                    // LAF: added
template void
GridUtils::normal_update(
  const Utils::DoFVector<2,2, Vector<double> > &N,
  const Utils::DoFVector<2,2, Vector<double> > &X);

template void
GridUtils::normal_update(
  const Utils::DoFVector<3,3, Vector<double> > &N,
  const Utils::DoFVector<3,3, Vector<double> > &X);



template class MoveMesh<1,2>;
template class MoveMesh<2,3>;

                    // LAF: added
template class MoveMesh<2,2>;
template class MoveMesh<3,3>;
```

Listing C.12 `make_graphs.m`

```
% ************************************************************
% File: make_graphs.m
% Author: Lauren Ferguson
% Created: August 2012
% Last updated: September 2012
%
% This file reads in the MATLAB data files created by the
% ConstST_Fracture.cc program and produces some nice
% graphics of the results. There are three series of results
% saved in the files: the crack profile, the slope u_{2,1}
% along the crack surface, and the stress component sigma_{22}
% along the bottom face outside the crack.
% ************************************************************

% A few new colors for the graphs.
ltb = [.2, .8, .8]; % Light blue
org = [1, .6, .2];  % Orange

% Legend positions
rect1 = [0.75, 0.71, 0.1, 0.1];
rect2 = [0.19, 0.25, 0.1, 0.1];
rect3 = [0.75, 0.75, 0.1, 0.1];
rect4 = [0.75, 0.72, 0.1, 0.1];
rect5 = [0.19, 0.20, 0.1, 0.1];

% Graph options:
```

```
% Choose graph_opt = 0 to just print the figures in the
% results files, which show convergence of the data sets
% across refinements.

% Graphs for fixed sigma:
% For sigma = 0.005, 0.025, 0.05, use options 1, 2, 3,
% respectively.

% Graphs for fixed gamma:
% gamma = 0.0, 0.001, 0.01, 0.1, 1.0, 10.0, use
% options 4, 5, 6, 7, 8, 9, respectively.

% Graphs for different domain sizes:
% Plots profiles for quadrant and bar for b=3,10,100 together,
% use option 10.

graph_opt = 1;

if (graph_opt == 0)
  Results_s0p05_g0p0;

  figure(2)
  set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
  set(h,'FontName','Arial','FontSize',20);
  set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
  ↳  ;
```

```
  set(h, 'Position', rect1);
end

X = 1:.0001:3;

if (graph_opt == 1)      % Graphs for fixed sigma = 0.005

  % Load results
  Results_s0p005_g0p0;
  Results_s0p005_g0p001;
  Results_s0p005_g0p01;
  Results_s0p005_g0p1;
  Results_s0p005_g1p0;
  Results_s0p005_g10p0;

  % Opening profile as gamma increases:
  figure(4);
  hold all;
  hnew = plot(P(:,1), P_s0p005_g0p0(:,1), 'b', 'Linewidth', 1.5);
  hnew = plot(P(:,1), P_s0p005_g0p001(:,1), 'g', 'Linewidth', 1.5)
  ↳  ;
  hnew = plot(P(:,1), P_s0p005_g0p01(:,1), 'r', 'Linewidth', 1.5);
  hnew = plot(P(:,1), P_s0p005_g0p1(:,1), 'Color', ltb, 'Linewidth
  ↳  ', 1.5);
  hnew = plot(P(:,1), P_s0p005_g1p0(:,1), 'm', 'Linewidth', 1.5);
  hnew = plot(P(:,1), P_s0p005_g10p0(:,1), 'Color', org, '
  ↳  Linewidth', 1.5);
```

```
h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
↳  gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
xlabel('Position on crack surface');
ylabel('Crack profile u_2');
title('Profile: sigma = 0.005');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
set(h, 'Position', rect1);

% Stress as gamma increases:
figure(5);
hold all;
hnew = plot(S(:,1), S_s0p005_g0p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p005_g0p001(:,1), 'g', 'Linewidth', 1.5)
↳  ;
hnew = plot(S(:,1), S_s0p005_g0p01(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p005_g0p1(:,1), 'Color', ltb, 'Linewidth
↳  ', 1.5);
hnew = plot(S(:,1), S_s0p005_g1p0(:,1), 'm', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p005_g10p0(:,1), 'Color', org, '
↳  Linewidth', 1.5);
h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
↳  gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
xlabel('Position on lower edge outside the crack');
```

```
ylabel('Stress sigma_{22}');
title('Stress: sigma = 0.005');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ;
set(h, 'Position', rect1);
axis([.999 1.025 -.25 4]);

% Slope as gamma increases:
figure(6);
hold all;
hnew = plot(U(:,1), U_s0p005_g0p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p005_g0p001(:,1), 'g', 'Linewidth', 1.5)
    ;
hnew = plot(U(:,1), U_s0p005_g0p01(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p005_g0p1(:,1), 'Color', ltb, 'Linewidth
    ', 1.5);
hnew = plot(U(:,1), U_s0p005_g1p0(:,1), 'm', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p005_g10p0(:,1), 'Color', org, '
    Linewidth', 1.5);
h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
    gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
xlabel('Position on the crack surface');
ylabel('Slope u_{2,1}');
title('Slope: sigma = 0.005');
```

```
120
    set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
    set(h,'FontName','Arial','FontSize',20);
    set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ↳  ;
    set(h, 'Position', rect2);
    axis([.975 1.001 -5 0.25]);

end



130 if (graph_opt == 2)      % Graphs for fixed sigma = 0.025

    % Load results
    Results_s0p025_g0p0;
    Results_s0p025_g0p001;
    Results_s0p025_g0p01;
    Results_s0p025_g0p1;
    Results_s0p025_g1p0;
    Results_s0p025_g10p0;

140   % Opening profile as gamma increases:
    figure(4);
    hold all;
    hnew = plot(P(:,1), P_s0p025_g0p0(:,1), 'b', 'Linewidth', 1.5);
    hnew = plot(P(:,1), P_s0p025_g0p001(:,1), 'g', 'Linewidth', 1.5)
    ↳  ;
```

```
 hnew = plot(P(:,1), P_s0p025_g0p01(:,1), 'r', 'Linewidth', 1.5);
 hnew = plot(P(:,1), P_s0p025_g0p1(:,1), 'Color', ltb, 'Linewidth
↳  ', 1.5);
 hnew = plot(P(:,1), P_s0p025_g1p0(:,1), 'm', 'Linewidth', 1.5);
 hnew = plot(P(:,1), P_s0p025_g10p0(:,1), 'Color', org, '
↳  Linewidth', 1.5);
 h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
↳  gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
 xlabel('Position on crack surface');
 ylabel('Crack profile u_2');
 title('Profile: sigma = 0.025');

 set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
 set(h,'FontName','Arial','FontSize',20);
 set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
 set(h, 'Position', rect1);

 % Stress as gamma increases:
 figure(5);
 hold all;
 hnew = plot(S(:,1), S_s0p025_g0p0(:,1), 'b', 'Linewidth', 1.5);
 hnew = plot(S(:,1), S_s0p025_g0p001(:,1), 'g', 'Linewidth', 1.5)
↳  ;
 hnew = plot(S(:,1), S_s0p025_g0p01(:,1), 'r', 'Linewidth', 1.5);
 hnew = plot(S(:,1), S_s0p025_g0p1(:,1), 'Color', ltb, 'Linewidth
↳  ', 1.5);
```

```
 hnew = plot(S(:,1), S_s0p025_g1p0(:,1), 'm', 'Linewidth', 1.5);
 hnew = plot(S(:,1), S_s0p025_g10p0(:,1), 'Color', org, '
↳  Linewidth', 1.5);
 h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
↳  gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
 xlabel('Position on lower edge outside the crack');
 ylabel('Stress sigma_{22}');
 title('Stress: sigma = 0.025');

 set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
 set(h,'FontName','Arial','FontSize',20);
 set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
 set(h, 'Position', rect1);
 axis([.999 1.025 -.25 5]);

 % Slope as gamma increases:
 figure(6);
 hold all;
 hnew = plot(U(:,1), U_s0p025_g0p0(:,1), 'b', 'Linewidth', 1.5);
 hnew = plot(U(:,1), U_s0p025_g0p001(:,1), 'g', 'Linewidth', 1.5)
↳  ;
 hnew = plot(U(:,1), U_s0p025_g0p01(:,1), 'r', 'Linewidth', 1.5);
 hnew = plot(U(:,1), U_s0p025_g0p1(:,1), 'Color', ltb, 'Linewidth
↳  ', 1.5);
 hnew = plot(U(:,1), U_s0p025_g1p0(:,1), 'm', 'Linewidth', 1.5);
```

```
  hnew = plot(U(:,1), U_s0p025_g10p0(:,1), 'Color', org, '
↳  Linewidth', 1.5);
  h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
↳  gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
  xlabel('Position on the crack surface');
  ylabel('Slope u_{2,1}');
  title('Slope: sigma = 0.025');

  set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
  set(h,'FontName','Arial','FontSize',20);
  set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
  set(h, 'Position', rect2);
  axis([.975 1.001 -5 0.25]);

end


if (graph_opt == 3)      % Graphs for fixed sigma = 0.05

  % Load results
  Results_s0p05_g0p0;
  Results_s0p05_g0p001;
  Results_s0p05_g0p01;
  Results_s0p05_g0p1;
  Results_s0p05_g1p0;
  Results_s0p05_g10p0;
```

```
% Opening profile as gamma increases:
figure(4);
hold all;
hnew = plot(P(:,1), P_s0p05_g0p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p05_g0p001(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p05_g0p01(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p05_g0p1(:,1), 'Color', ltb, 'Linewidth'
↳ , 1.5);
hnew = plot(P(:,1), P_s0p05_g1p0(:,1), 'm', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p05_g10p0(:,1), 'Color', org, 'Linewidth
↳ ', 1.5);
h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
↳ gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
xlabel('Position on crack surface');
ylabel('Crack profile u_2');
title('Profile: sigma = 0.05');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳ ;
set(h, 'Position', rect1);

% Stress as gamma increases:
figure(5);
hold all;
```

```
 hnew = plot(S(:,1), S_s0p05_g0p0(:,1), 'b', 'Linewidth', 1.5);
 hnew = plot(S(:,1), S_s0p05_g0p001(:,1), 'g', 'Linewidth', 1.5);
 hnew = plot(S(:,1), S_s0p05_g0p01(:,1), 'r', 'Linewidth', 1.5);
 hnew = plot(S(:,1), S_s0p05_g0p1(:,1), 'Color', ltb, 'Linewidth'
↳  , 1.5);
 hnew = plot(S(:,1), S_s0p05_g1p0(:,1), 'm', 'Linewidth', 1.5);
 hnew = plot(S(:,1), S_s0p05_g10p0(:,1), 'Color', org, 'Linewidth
↳  ', 1.5);
240 h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
↳  gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
 xlabel('Position on lower edge outside the crack');
 ylabel('Stress sigma_{22}');
 title('Stress: sigma = 0.05');

 set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
 set(h,'FontName','Arial','FontSize',20);
 set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
 set(h, 'Position', rect1);
 axis([.999 1.025 -.25 9]);
250

 % Slope as gamma increases:
 figure(6);
 hold all;
 hnew = plot(U(:,1), U_s0p05_g0p0(:,1), 'b', 'Linewidth', 1.5);
 hnew = plot(U(:,1), U_s0p05_g0p001(:,1), 'g', 'Linewidth', 1.5);
 hnew = plot(U(:,1), U_s0p05_g0p01(:,1), 'r', 'Linewidth', 1.5);
```

```
 hnew = plot(U(:,1), U_s0p05_g0p1(:,1), 'Color', ltb, 'Linewidth'
↳   , 1.5);
 hnew = plot(U(:,1), U_s0p05_g1p0(:,1), 'm', 'Linewidth', 1.5);
 hnew = plot(U(:,1), U_s0p05_g10p0(:,1), 'Color', org, 'Linewidth
↳   ', 1.5);
 h = legend('gamma = 0.0', 'gamma = 0.001', 'gamma = 0.01', '
↳  gamma = 0.1', 'gamma = 1.0', 'gamma = 10.0');
 xlabel('Position on the crack surface');
 ylabel('Slope u_{2,1}');
 title('Slope: sigma = 0.05');

 set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
 set(h,'FontName','Arial','FontSize',20);
 set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
 set(h, 'Position', rect2);
 axis([.955 1.001 -6 0.25]);

end



if (graph_opt == 4)      % Graphs for fixed gamma = 0.0

 % Load results
 Results_s0p005_g0p0;
 Results_s0p025_g0p0;
 Results_s0p05_g0p0;
```

```
280

% Opening profile as sigma increases:
figure(7);
hold all;
hnew = plot(P(:,1), P_s0p005_g0p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p025_g0p0(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p05_g0p0(:,1), 'r', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
xlabel('Position on crack surface');
ylabel('Crack profile u_2');
290 title('Profile: gamma = 0.0');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳ ;
set(h, 'Position', rect3);

% Stress as sigma increases:
figure(8);
hold all;
300 hnew = plot(S(:,1), S_s0p005_g0p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p025_g0p0(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p05_g0p0(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(X, 1./sqrt(X-1), 'Color', ltb, 'Linewidth', 1.5);
hnew = plot(X, -log((X-1)), 'm', 'Linewidth', 1.5);
```

```
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05', '1/
↳  sqrt(x-1)', '-log[(x-1)]');
xlabel('Position on lower edge outside the crack');
ylabel('Stress sigma_{22}');
title('Stress: gamma = 0.0');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
set(h, 'Position', rect4);
axis([.999 1.025 -.25 125]);

% Slope as sigma increases:
figure(9);
hold all;
hnew = plot(U(:,1), U_s0p005_g0p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p025_g0p0(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p05_g0p0(:,1), 'r', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
xlabel('Position on the crack surface');
ylabel('Slope u_{2,1}');
title('Slope: gamma = 0.0');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
```

```
  set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳   ;
  set(h, 'Position', rect5);
  axis([.95 1.001 -250 5]);

end



if (graph_opt == 5)      % Graphs for fixed gamma = 0.001

  % Load results
  Results_s0p005_g0p001;
  Results_s0p025_g0p001;
  Results_s0p05_g0p001;

  % Opening profile as sigma increases:
  figure(7);
  hold all;
  hnew = plot(P(:,1), P_s0p005_g0p001(:,1), 'b', 'Linewidth', 1.5)
↳   ;
  hnew = plot(P(:,1), P_s0p025_g0p001(:,1), 'g', 'Linewidth', 1.5)
↳   ;
  hnew = plot(P(:,1), P_s0p05_g0p001(:,1), 'r', 'Linewidth', 1.5);
  h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
  xlabel('Position on crack surface');
  ylabel('Crack profile u_2');
  title('Profile: gamma = 0.001');
```

```
set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ;
set(h, 'Position', rect3);

% Stress as sigma increases:
figure(8);
hold all;
hnew = plot(S(:,1), S_s0p005_g0p001(:,1), 'b', 'Linewidth', 1.5)
    ;
hnew = plot(S(:,1), S_s0p025_g0p001(:,1), 'g', 'Linewidth', 1.5)
    ;
hnew = plot(S(:,1), S_s0p05_g0p001(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(X, 1./sqrt(X-1), 'Color', ltb, 'Linewidth', 1.5);
hnew = plot(X, -log((X-1)), 'm', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05', '1/
    sqrt(x-1)', '-log[(x-1)]');
xlabel('Position on lower edge outside the crack');
ylabel('Stress sigma_{22}');
title('Stress: gamma = 0.001');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ;
```

```
  set(h, 'Position', rect4);
  axis([.999 1.025 -.25 40]);

  % Slope as sigma increases:
  figure(9);
  hold all;
  hnew = plot(U(:,1), U_s0p005_g0p001(:,1), 'b', 'Linewidth', 1.5)
  ;
  hnew = plot(U(:,1), U_s0p025_g0p001(:,1), 'g', 'Linewidth', 1.5)
  ;
  hnew = plot(U(:,1), U_s0p05_g0p001(:,1), 'r', 'Linewidth', 1.5);
  h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
  xlabel('Position on the crack surface');
  ylabel('Slope u_{2,1}');
  title('Slope: gamma = 0.001');

  set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
  set(h,'FontName','Arial','FontSize',20);
  set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
  ;
  set(h, 'Position', rect5);
  axis([.95 1.001 -5.5 0.25]);

end



if (graph_opt == 6)      % Graphs for fixed gamma = 0.01
```

```
% Load results
Results_s0p005_g0p01;
Results_s0p025_g0p01;
Results_s0p05_g0p01;

% Opening profile as sigma increases:
figure(7);
hold all;
hnew = plot(P(:,1), P_s0p005_g0p01(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p025_g0p01(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p05_g0p01(:,1), 'r', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
xlabel('Position on crack surface');
ylabel('Crack profile u_2');
title('Profile: gamma = 0.01');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
set(h, 'Position', rect3);

% Stress as sigma increases:
figure(8);
hold all;
hnew = plot(S(:,1), S_s0p005_g0p01(:,1), 'b', 'Linewidth', 1.5);
```

```
hnew = plot(S(:,1), S_s0p025_g0p01(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p05_g0p01(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(X, 1./sqrt(X-1), 'Color', ltb, 'Linewidth', 1.5);
hnew = plot(X, -log((X-1)), 'm', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05', '1/
↳  sqrt(x-1)', '-log[(x-1)]');
xlabel('Position on lower edge outside the crack');
ylabel('Stress sigma_{22}');
title('Stress: gamma = 0.01');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
set(h, 'Position', rect4);
axis([.999 1.025 -.25 40]);

% Slope as sigma increases:
figure(9);
hold all;
hnew = plot(U(:,1), U_s0p005_g0p01(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p025_g0p01(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p05_g0p01(:,1), 'r', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
xlabel('Position on the crack surface');
ylabel('Slope u_{2,1}');
title('Slope: gamma = 0.01');
```

```
450
    set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
    set(h,'FontName','Arial','FontSize',20);
    set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ↳  ;
    set(h, 'Position', rect5);
    axis([.95 1.001 -2.5 0.25]);

end



460 if (graph_opt == 7)      % Graphs for fixed gamma = 0.1

    % Load results
    Results_s0p005_g0p1;
    Results_s0p025_g0p1;
    Results_s0p05_g0p1;

    % Opening profile as sigma increases:
    figure(7);
    hold all;
470 hnew = plot(P(:,1), P_s0p005_g0p1(:,1), 'b', 'Linewidth', 1.5);
    hnew = plot(P(:,1), P_s0p025_g0p1(:,1), 'g', 'Linewidth', 1.5);
    hnew = plot(P(:,1), P_s0p05_g0p1(:,1), 'r', 'Linewidth', 1.5);
    h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
    xlabel('Position on crack surface');
    ylabel('Crack profile u_2');
```

```
title('Profile: gamma = 0.1');


set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ;
set(h, 'Position', rect3);

% Stress as sigma increases:
figure(8);
hold all;
hnew = plot(S(:,1), S_s0p005_g0p1(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p025_g0p1(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p05_g0p1(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(X, 1./sqrt(X-1), 'Color', ltb, 'Linewidth', 1.5);
hnew = plot(X, -log((X-1)), 'm', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05', '1/
    sqrt(x-1)', '-log[(x-1)]');
xlabel('Position on lower edge outside the crack');
ylabel('Stress sigma_{22}');
title('Stress: gamma = 0.1');


set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ;
set(h, 'Position', rect4);
```

```
  axis([.999 1.025 -.25 40]);

  % Slope as sigma increases:
  figure(9);
  hold all;
  hnew = plot(U(:,1), U_s0p005_g0p1(:,1), 'b', 'Linewidth', 1.5);
  hnew = plot(U(:,1), U_s0p025_g0p1(:,1), 'g', 'Linewidth', 1.5);
  hnew = plot(U(:,1), U_s0p05_g0p1(:,1), 'r', 'Linewidth', 1.5);
  h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
  xlabel('Position on the crack surface');
  ylabel('Slope u_{2,1}');
  title('Slope: gamma = 0.1');

  set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
  set(h,'FontName','Arial','FontSize',20);
  set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
  ↳ ;
  set(h, 'Position', rect5);
  axis([.95 1.001 -1.5 0.25]);

end


if (graph_opt == 8)      % Graphs for fixed gamma = 1.0

  % Load results
  Results_s0p005_g1p0;
```

```
Results_s0p025_g1p0;
Results_s0p05_g1p0;

% Opening profile as sigma increases:
figure(7);
hold all;
hnew = plot(P(:,1), P_s0p005_g1p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p025_g1p0(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(P(:,1), P_s0p05_g1p0(:,1), 'r', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
xlabel('Position on crack surface');
ylabel('Crack profile u_2');
title('Profile: gamma = 1.0');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
set(h, 'Position', rect3);

% Stress as sigma increases:
figure(8);
hold all;
hnew = plot(S(:,1), S_s0p005_g1p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p025_g1p0(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p05_g1p0(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(X, 1./sqrt(X-1), 'Color', ltb, 'Linewidth', 1.5);
```

```
hnew = plot(X, -log((X-1)), 'm', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05', '1/
↳  sqrt(x-1)', '-log[(x-1)]');
xlabel('Position on lower edge outside the crack');
ylabel('Stress sigma_{22}');
title('Stress: gamma = 1.0');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
set(h, 'Position', rect4);
axis([.999 1.025 -.25 40]);

% Slope as sigma increases:
figure(9);
hold all;
hnew = plot(U(:,1), U_s0p005_g1p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p025_g1p0(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(U(:,1), U_s0p05_g1p0(:,1), 'r', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
xlabel('Position on the crack surface');
ylabel('Slope u_{2,1}');
title('Slope: gamma = 1.0');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
```

```
  set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳   ;
  set(h, 'Position', rect5);
  axis([.95 1.001 -1.5 0.25]);

end



if (graph_opt == 9)      % Graphs for fixed gamma = 10.0

  % Load results
  Results_s0p005_g10p0;
  Results_s0p025_g10p0;
  Results_s0p05_g10p0;

  % Opening profile as sigma increases:
  figure(7);
  hold all;
  hnew = plot(P(:,1), P_s0p005_g10p0(:,1), 'b', 'Linewidth', 1.5);
  hnew = plot(P(:,1), P_s0p025_g10p0(:,1), 'g', 'Linewidth', 1.5);
  hnew = plot(P(:,1), P_s0p05_g10p0(:,1), 'r', 'Linewidth', 1.5);
  h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
  xlabel('Position on crack surface');
  ylabel('Crack profile u_2');
  title('Profile: gamma = 10.0');

  set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
```

```
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ;
set(h, 'Position', rect3);

% Stress as sigma increases:
figure(8);
hold all;
hnew = plot(S(:,1), S_s0p005_g10p0(:,1), 'b', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p025_g10p0(:,1), 'g', 'Linewidth', 1.5);
hnew = plot(S(:,1), S_s0p05_g10p0(:,1), 'r', 'Linewidth', 1.5);
hnew = plot(X, 1./sqrt(X-1), 'Color', ltb, 'Linewidth', 1.5);
hnew = plot(X, -log((X-1)), 'm', 'Linewidth', 1.5);
h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05', '1/
    sqrt(x-1)', '-log[(x-1)]');
xlabel('Position on lower edge outside the crack');
ylabel('Stress sigma_{22}');
title('Stress: gamma = 10.0');

set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
set(h,'FontName','Arial','FontSize',20);
set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
    ;
set(h, 'Position', rect4);
axis([.999 1.025 -.25 40]);

% Slope as sigma increases:
```

```
  figure(9);
  hold all;
  hnew = plot(U(:,1), U_s0p005_g10p0(:,1), 'b', 'Linewidth', 1.5);
  hnew = plot(U(:,1), U_s0p025_g10p0(:,1), 'g', 'Linewidth', 1.5);
  hnew = plot(U(:,1), U_s0p05_g10p0(:,1), 'r', 'Linewidth', 1.5);
  h = legend('sigma = 0.005', 'sigma = 0.025', 'sigma = 0.05');
  xlabel('Position on the crack surface');
  ylabel('Slope u_{2,1}');
  title('Slope: gamma = 10.0');

  set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
  set(h,'FontName','Arial','FontSize',20);
  set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
  ;
  set(h, 'Position', rect5);
  axis([.95 1.001 -.05 0.005]);

end



if (graph_opt == 10)      % Graphs for different domain sizes

  % Load results
  Results_b3_s0p05_g0p0;        % bar, b = 3
  Results_b3_s0p05_g1p0;
  Results_b10_s0p05_g0p0;       % bar, b = 10
  Results_b10_s0p05_g1p0;
```

```
Results_b100_s0p05_g0p0;      % bar, b = 100
Results_b100_s0p05_g1p0;

Results_q3_s0p05_g0p0;        % quadrant, b = 3
Results_q3_s0p05_g1p0;
Results_q10_s0p05_g0p0;       % quadrant, b = 10
Results_q10_s0p05_g1p0;
Results_q100_s0p05_g0p0;      % quadrant, b = 100
Results_q100_s0p05_g1p0;

% Opening profile as b increases for fixed gamma = 0.0:
figure(10);
hold all;

idx3 = 1:19:length(P(:,1));
idx10 = 7:19:length(P(:,1));
idx100 = 13:19:length(P(:,1));
xb3 = P(idx3,1);
xb10 = P(idx10,1);
xb100 = P(idx100,1);
yb3 = P10_b3_s0p05_g0p0(idx3,1);
yb10 = P10_b10_s0p05_g0p0(idx10,1);
yb100 = P10_b100_s0p05_g0p0(idx100,1);
hnew = plot(xb3, yb3,'b*','MarkerSize', 10.0);
hnew = plot(xb10, yb10,'gs','MarkerSize', 10.0);
hnew = plot(xb100, yb100,'ro','MarkerSize', 10.0);
```

```
680  hnew = plot(P(:,1), P10_q3_s0p05_g0p0(:,1), 'm', 'Linewidth',
↳  1.5);
 hnew = plot(P(:,1), P10_q10_s0p05_g0p0(:,1), 'Color', ltb, '
↳  Linewidth', 1.5);
 hnew = plot(P(:,1), P10_q100_s0p05_g0p0(:,1), 'Color', org, '
↳  Linewidth', 1.5);
 hnew = plot(P(:,1), P10_b3_s0p05_g0p0(:,1), 'b', 'Linewidth',
↳  1.5);
 hnew = plot(P(:,1), P10_b10_s0p05_g0p0(:,1), 'g', 'Linewidth',
↳  1.5);
 hnew = plot(P(:,1), P10_b100_s0p05_g0p0(:,1), 'r', 'Linewidth',
↳  1.5);

 h = legend('B3', 'B10', 'B100', 'Q3', 'Q10', 'Q100');
 xlabel('Position on crack surface');
 ylabel('Crack profile u_2');
690  title('Profile: sigma = 0.05, gamma = 0.0');

 set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
 set(h,'FontName','Arial','FontSize',20);
 set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
↳  ;
 set(h, 'Position', rect1);

 % Opening profile as b increases for fixed gamma = 1.0:
 figure(11);
 hold all;
```

```
700
    yb3 = P10_b3_s0p05_g1p0(idx3,1);
    yb10 = P10_b10_s0p05_g1p0(idx10,1);
    yb100 = P10_b100_s0p05_g1p0(idx100,1);
    hnew = plot(xb3, yb3,'b*','MarkerSize', 10.0);
    hnew = plot(xb10, yb10,'gs','MarkerSize', 10.0);
    hnew = plot(xb100, yb100,'ro','MarkerSize', 10.0);

    hnew = plot(P(:,1), P10_q3_s0p05_g1p0(:,1), 'm', 'Linewidth',
   ↳  1.5);
    hnew = plot(P(:,1), P10_q10_s0p05_g1p0(:,1), 'Color', ltb, '
   ↳  Linewidth', 1.5);
710 hnew = plot(P(:,1), P10_q100_s0p05_g1p0(:,1), 'Color', org, '
   ↳  Linewidth', 1.5);
    hnew = plot(P(:,1), P10_b3_s0p05_g1p0(:,1), 'b', 'Linewidth',
   ↳  1.5);
    hnew = plot(P(:,1), P10_b10_s0p05_g1p0(:,1), 'g', 'Linewidth',
   ↳  1.5);
    hnew = plot(P(:,1), P10_b100_s0p05_g1p0(:,1), 'r', 'Linewidth',
   ↳  1.5);

    h = legend('B3', 'B10', 'B100', 'Q3', 'Q10', 'Q100');
    xlabel('Position on crack surface');
    ylabel('Crack profile u_2');
    title('Profile: sigma = 0.05, gamma = 1.0');

720 set(get(gcf,'CurrentAxes'),'FontName','Arial','FontSize',14);
```

```
  set(h,'FontName','Arial','FontSize',20);
  set(findall(gca,'type','text'),'FontName','Arial','FontSize',20)
 ↳  ;
  set(h, 'Position', rect1);
end
```

APPENDIX D

MATHEMATICA® CODE FOR COMPUTING THE L^2 NORM OF THE KERNEL

The following listings contain the Mathematica® (Wolfram Research, Inc. 2010) files that we created to compute the estimated L^2 norm of the kernel, which is the lower bound on the absolute value of the second (nondimensional) surface tension parameter γ_1. They consist of:

D.1 `KernelNorm.nb` The main notebook that calls the function to compute the L^2 norm of the kernel and output the results.

D.2 `ComputeL2Norm.m` A package that provides the functions `computeNormSplit` and `computeNorm` which do the heavy lifting to compute the L^2 norm of the kernel.

D.3 `kernel.m` A package that provides the function `kernel` which computes the value of the kernel for a given pair $(x, q) \in S$.

D.4 `kernel_funcs.m` A package that defines the continuous integral functions from Table 4.6 that appear in the `kernel` function and are computed for each $(x, q) \in S$ using Gaussian quadrature.

D.5 `kernel_test.nb` A short test program that checks that the `kernel` function is working correctly.

Listing D.1 `KernelNorm.nb`

```
(* ************************************************************
File: KernelNorm.nb
Author: Lauren Ferguson
Created: June 2012
Last updated: June 2012

This Mathematica notebook is the main driver for computing the
L2 norm of the kernel k(x,q). It calls the routine that actually
computes this norm (either computeNorm or computeNormSplit, both
found in the package file "ComputeL2Norm.m") given the following:

 Inputs:
    nu = Poisson's ratio for the desired material
    gamma0 = the first (nondimensional) surface tension parameter
    nNodes = number of nodes along the interval (0,1].

The main output is the L2 norm:

Outputs:
    norm (or splitNorm) = L2 norm of the interpolant of the
                          kernel k(x,q)

This routine also writes a summary of the input/output data and
appends it to the file "NormResults.ml".
************************************************************ *)
```

```
(* First, change the working directory to the directory containing
   this notebook. *)
SetDirectory[NotebookDirectory[]];

(* Then load the package containing the routines to compute the
   L2 norm. *)
Get["ComputeL2Norm.m"]

(* Next, define the inputs. *)
nu = 0.2315;
gamma0 = 0.001;
nNodes = 300;

(* As discussed in ComputeL2Norm.m, there are two functions we can
   call to compute the norm: computeNorm and computeNormSplit.
   They do the exact same thing, but the split version seems to be
   faster for a large number of nodes, so we typically call that
   version. *)
(* norm = computeNorm[nu,gamma0,nNodes] *)
norm = computeNormSplit[nu, gamma0, nNodes]

(* Finally, we print a summary of the results to
   "NormResults.ml". *)
Clear[myout];
myout = OpenAppend["NormResults.ml"];
WriteString[myout, "***** NormResults.m1 *****\n\n"];
```

```
WriteString[myout, "Input: \n"];
WriteString[myout, "    nu: ", nu, "\n"];
WriteString[myout, "    gamma0: ", gamma0, "\n"];
WriteString[myout, "    nNodes: ", nNodes, "\n"];

WriteString[myout, "Output: \n"];
WriteString[myout, "    total nodes: ", totalNodes, "\n"];
WriteString[myout, "    step size: ", stepSize, "\n"];
WriteString[myout, "    time taken: ", timeTaken, "\n"];
WriteString[myout, "    L2 norm: "];
Write[myout, norm]
WriteString[myout, "\n"];

(* Finally, append a tex line that summarizes the data for a table
   entry. *)

WriteString[myout,
  "gamma0 & nNodes & totalNodes & stepSize & L2 norm \\\\ \n"];
WriteString[myout, gamma0, " & " ];
WriteString[myout, nNodes, " & "];
WriteString[myout, totalNodes, " & "];
WriteString[myout, stepSize, " & "];
WriteString[myout, OutputForm[N[norm, 10]], " \\\\ "];
WriteString[myout, "\n\n\n"];

Close[myout];
```

Listing D.2 `ComputeL2Norm.m`

```
(* ************************************************************
File: ComputeL2Norm.m
Author: Lauren Ferguson
Created: June 2012
Last updated: June 2012

This package defines two functions for computing the L2 norm of
the kernel k(x,q), which is defined in the package "kernel.m".
The norm is taken over the domain S = {(x,q) ∈ [-1,1]^2 : x ≠ q},
since we use the fact that the set D = {(x,q) ∈ [-1,1]^2 : x = q}
is a set of measure zero.

In the first function (computeNormSplit), we split the integral
into upper and lower triangles and set the kernel equal to zero
on D and on the opposite triangle to obtain the desired result.
In other words, we compute
Sqrt[\int_-1^1\int_-1^1 k_1(x,q)dxdq
     + \int_-1^1\int_-1^1 k_2(x,q)dxdq],
where k_1 is a cubic spline interpolant on the square domain
[-1,1]^2 whose values at the nodes on the upper triangle are
computed by (kernel[x,q])^2 and whose values on nodes of the
lower triangle and the diagaonal are identically zero. (We do
this because the interpolation routine requires a square
domain.) Similarly, k_2 is a cubic spline interpolant on the
square domain that takes the value zero on the upper triangle
and the diagonal and the value computed by (kernel[x,q])^2 on
```

```
the lower triangle.

The second function (computeNorm) does the same thing without
the split. It defines a cubic spline interpolant over the whole
square [-1,1]^2 whose values at the nodes are computed by
(kernel[x,q])^2, except along the diagonal, where it takes the
values to be zero. Then it computes the norm
Sqrt[\int_-1^1\int_-1^1 (k(x,q))^2 dxdq]

The main advantage of the split routine is that it takes less
time, which is especially useful if a large number of nodes is
used. It seems that the value of the norm is the same when
computed from either routine.

For both functions, we have:

Inputs:
    nu = Poisson's ratio for the desired material
    gamma0 = the first (nondimensional) surface tension parameter
    nNodes = number of nodes along the interval (0,1]. This is
             really only used to define the stepsize and the total
             number of nodes.
Outputs:
    norm = L2 norm of the interpolant of the kernel k (x,q)
********************************************************** *)

(* Both functions require the packages "kernel.m" and
```

```
    "kernel_funcs.m" to be loaded. These define the kernel and the
    numerically integrated functions required to compute the
    kernel. *)
Get["kernel_funcs.m"];
Get["kernel.m"];

computeNormSplit[in1_,in2_,in3_]:=
  Module[{nu = in1, gamma0 = in2, nNodes = in3, norm},

  (* First, compute the stepSize and some info about the nodes: *)
  stepSize = N[1/nNodes];
  nodesPerSide = nNodes*2 + 1; (* nodes per side of the square *)
  totalNodes = nodesPerSide^2; (* total number of nodes *)

  (* We also keep track of the time taken to find the L2 norm.
     With this data, we clearly see the advantage to the split
     routine over the non-split version. *)
  time1 = AbsoluteTime[];

  (* Next, we define the interpolant of the square of the kernel
     on the upper triangle of the domain. We collect the data
     points for this interpolation into the table interpDataA.
     Then we find the interpolant with the Interpolation routine,
     using cubic splines, and store the result in kernelInterpA.
     The Interpolation routine requires a square table, so we have
     actually computed the interpolant over the whole domain,
     where we have set the value of the kernel squared in the
```

```
   lower triangle and on the diagonal to zero. *)
interpDataA = Flatten[Table[{{x,q},
                If[x < q,(kernel[x,q,nu,gamma0])^2,0]},
                {x,-1,1,stepSize},{q,-1,1,stepSize}],1];
kernelInterpA = Interpolation[interpDataA,
                  Method->"Spline",InterpolationOrder-> 3];

(* Then the double integral of the interpolant is: *)
intA = NIntegrate[kernelInterpA[x,q],{x,-1,1},{q,-1,1},
          MaxRecursion->10000];
(* Since we only need intA, we clear the interpolation
   variables to save space. *)
clear[interpDataA, kernelInterpA];

(* Similarly, we apply the same process to find the interpolant
   and double integral for the lower triangle of the domain. *)
interpDataB = Flatten[Table[{{x,q},
                If[x > q,(kernel[x,q,nu,gamma0])^2,0]},
                {x,-1,1,stepSize},{q,-1,1,stepSize}],1];
kernelInterpB = Interpolation[interpDataB,
                  Method->"Spline",InterpolationOrder-> 3];
intB = NIntegrate[kernelInterpB[x,q],{x,-1,1},{q,-1,1},
          MaxRecursion->10000];
clear[interpDataB, kernelInterpB];

(* Finally, we compute the L2 norm by taking the square root
   of the sum of the upper and lower integrals. *)
```

```
  norm = Sqrt[intA + intB];

  (* We also output the total time taken.*)
  time2 = AbsoluteTime[];
  totalTime = time2 - time1;
  timeH = Quotient[totalTime, 3600];
  timeM = Quotient[totalTime - timeH*3600, 60];
  timeS = Floor[totalTime - timeH*3600 - timeM*60];
  timeTaken := StringJoin[
    PadLeft[Characters[ToString[timeH]], 2, "0"], ":",
    PadLeft[Characters[ToString[timeM]], 2, "0"], ":",
    PadLeft[Characters[ToString[timeS]], 2, "0"]];

  (* Finally, return the value of the norm. *)
  N[norm]
]
computeNormSplit::usage = "computeNormSplit usage:
L2norm = computeNormSplit[nu, gamma0, and nNodes]
Similarly for computeNorm."

computeNorm[in1_,in2_,in3_]:=
  Module[{nu = in1, gamma0 = in2, nNodes = in3, norm},

  (* First, compute the step size and some info about the
     nodes: *)
  stepSize = N[1/nNodes];
  nodesPerSide = nNodes*2 + 1; (* nodes per side of the square *)
```

```
totalNodes = nodesPerSide^2; (* total number of nodes *)

(* We also keep track of the time taken to find the L2 norm. *)
time1 = AbsoluteTime[];

(* Next, we define the interpolant of the square of the kernel.
   We collect the data points for the interpolation into the
   table interpData. Then we find the interpolant with the
   Interpolation routine using cubic splines. *)
xqPoints={};
For[i = 0, i < nodesPerSide, i++,
  AppendTo[xqPoints, -1+i*stepSize]];

interpData = Flatten[Table[{{x,q},(kernel[x,q,nu,gamma0])^2},
                {x,xqPoints},{q,xqPoints}],1];
kernelInterp=Interpolation[interpData,Method->"Spline",
                InterpolationOrder->3];

(* Finally, we find the double integral of the interpolant, and
   take the square root to obtain the L2 norm. *)
kernelInt = NIntegrate[kernelInterp[x,q],{x,-1,1},{q,-1,1},
              MaxRecursion->10000];
clear[interpData, kernelInterp];
norm = Sqrt[kernelInt];

(* We also output the total time taken.*)
time2 = AbsoluteTime[];
```

```
  totalTime = time2 - time1;
  timeH = Quotient[totalTime, 3600];
  timeM = Quotient[totalTime - timeH*3600, 60];
  timeS = Floor[totalTime - timeH*3600 - timeM*60];
  timeTaken := StringJoin[
    PadLeft[Characters[ToString[timeH]], 2, "0"], ":",
    PadLeft[Characters[ToString[timeM]], 2, "0"], ":",
    PadLeft[Characters[ToString[timeS]], 2, "0"]];

  (* Finally, return the value of the norm. *)
  N[norm]
]
computeNorm::usage = "computeNorm usage:
L2norm = computeNorm[nu, gamma0, and nNodes]
Similarly for computeNormSplit."
```

Listing D.3 `kernel.m`

```
(* ************************************************************
File: kernel.m
Author: Lauren Ferguson
Created: July 2010
Last Updated: June 2012

This package provides the routine kernel[x,q] which computes the
value of the kernel k(x,q) at any point (x,q) in the domain
S = {(x,q) ∈ [-1,1]^2 : x ≠ q}. It is used in conjunction
with the routines in the package "ComputeL2norm.m" which compute
the L2 norm of this kernel. As discussed in "ComputeL2norm.m",
this function returns a value of zero for points on the set
D = {(x,q): -1 ≤ x = q ≤ 1}. The kernel function also requires
the library of functions "kernel_funcs.m" to be loaded. These
represent all the integrals in the kernel that must be computed
via numerical integration.

Inputs:
    x = x-value of the point. (-1 ≤ x ≤ 1)
    q = q-value of the point. (-1 ≤ q ≤ 1)
    nu = Poisson's ratio
    gamma0 = first (nondimensional) surface tension parameter

We also define a flag that determines whether or not to print some
data. In general, we set this flag to zero (for no output). This
statement should be commented out when testing the kernel using
```

```
the notebook "kernel_test.nb".
    outputFlag = bool flag for output
********************************************************** *)

kernel[in1_,in2_,in3_,in4_]:=
  Module[{x = in1, q = in2, nu = in3, gamma0 = in4},

  outputFlag = 0;
  (* In general, Poisson's ratio has the range -1 < nu ≤ 0.5,
     but only attains the value 0.5 if the material is
     incompressible, which we will avoid here. *)
  If[((nu ≥ 0.5) || (nu ≤ -1)),
    Print["Error: nu-value is out of range"]; Abort[]];

  zeta1star = 1/(2(1-nu));
  zeta2 = (1-2nu)zeta1star;
  c1 = -gamma0/(zeta2*Pi);
  c2 = zeta1star/(zeta2*Pi^2);

  (* If[outputFlag ≠ 0,
    Print["Evaluating kernel with nu: ", nu, ", gamma0: ",
          gamma0],0]; *)

  abx = Abs[x];
  abq = Abs[q];

  (* Check to see that (x, q) is in the domain [-1,1]^2 *)
```

```
If[abx > 1, Print["Error: x-value is out of range"]; Abort[]];
If[abq > 1, Print["Error: q-value is out of range"]; Abort[]];

(* ********** Term 1 ********** *)
(* T1 = c1*Sqrt[1-x^2]*I1, where
   I1 = \cpv_q^1 \frac{dr}{Sqrt[1 - r^2](r - x)} *)
T1 = 0;
I1 = 0;

Which[((abx < 1) && (abq < 1)),
  Which[x < q,
    If[outputFlag == 1,Print["Case 1.5: x < q ∈ (-1, 1)"]];
    I1 = Pi/(2(1-x)) - ArcSin[q]/(q-x)+I1C6[x,q];
    T1 = c1*Sqrt[1-x^2]I1;
   ,x > q,
    If[outputFlag == 1,Print["Case 1.6: x > q ∈ (-1, 1)"]];
    T1 = Sqrt[1-x]/(Sqrt[2]+Sqrt[1+x])*(-2*Sqrt[2]*f1a[x,q]
         + f1b[x,q]);
    T1 += (1-x)/(Sqrt[2]+Sqrt[1+x])*(Log[1-x]-Log[x-q]);
    T1 += Sqrt[1+x]*(2*Log[Sqrt[1-q]+Sqrt[1-x]]-Log[x-q]);
    T1 *= c1/Sqrt[2];
   ,True, (* else *)
    If[outputFlag == 1, Print["Case 1.8: x = q ∈ (-1, 1)"]];
    (* Print["Set diagonal to zero."]; *)
    T1 = 0;
  ]
 ,((x == 1) && (q ≠ 1)),
```

```
    If[outputFlag == 1,Print["Case 1.4: x=1"]];
    T1 = -2*c1;
  ,True,
    If[outputFlag == 1,Print["Case 1.1, 1.2, 1.3, or 1.7"]];
    T1 = 0;
  ];
  If[outputFlag == 1,Print["T1 = ", N[T1]]];

  (* ************ Term 2 ************ *)
  (* T2 = c2*Sqrt[1-x^2]\cpv_-1^1\frac{q-r}{Sqrt[1-r^2](r-x)}dr *)
  T2 = 0;

  Which[((x == -1) && (q ≠ -1)),
    If[outputFlag == 2, Print["Case 2.1: x = -1, q ≠ -1"]];
    T2 = 2*c2*(1+q);
  ,((x == 1) && (q ≠ 1)),
    If[outputFlag == 2,Print["Case 2.2: x = 1, q ≠ 1"]];
    T2 = 2*c2*(1-q);
  ,(x ≠ q),
    If[outputFlag == 2, Print["Case 2.3: x ∈ (-1,1), q ≠ x"]];
    T2 = -c2*Pi Sqrt[1-x^2];
  ,True, (* else *)
    If[outputFlag == 2, Print["Case 2.4: x = q ∈ [-1,1]"]];
    T2 = 0;
  ];
  If[outputFlag == 2, Print["T2 = ", N[T2]]];
```

```
(* ************ Term 3 ************ *)
(* T3 = c2*Sqrt[1-x^2]\cpv_-1^1\frac{(q-r)Log[Abs[q-r]]}
        {Sqrt[1-r^2](r-x)}dr
     = c2 Sqrt[1-x^2](-g1(q) + (q-x)g2(x,q)), where
   g1(q) = \cpv_-1^1\frac{Log[Abs[q-r]]}{Sqrt[1-r^2]}dr, and
   g2(x,q) = \cpv_-1^1\frac{Log[Abs[q-r]]}
            {Sqrt[1-r^2](r-x)}dr *)
T3 = 0;

(* g1(q) is the same for all q: *)
g1 = -Pi*Log[2];
(* If[outputFlag == 3, Print["g1 = ", g1]]; *)

(* g2(x,q): Only needs to be computed for x ≠ q ∈ [-1,1]^2,
   and we also compute g2star = Sqrt[1-x^2]g2(x,q) *)
g2 = 0;
g2star = 0;

If[x == q,
  T3 = 0; (* No contribution on the diagonal *)
, (* else *)
  Which[((abq < 1) && (x ≠ 0) && (abx < 1)),
    If[outputFlag == 3, Print["Case 3.1: x ∈ (-1,1)\{0}, q ≠ x,
      or equiv q = 0"]];
    (* Compute g2(abx, 0) *)
    g2 = -2/abx*ArcSin[abx/2]Log[abx/2]
         -2/(1-abx)ArcSin[(1+abx)/2]Log[(1+abx)/2];
```

```
    g2 += Log[abx]/Sqrt[1-abx^2] (g2C1d[abx]
          + Log[Abs[1-abx]]-Log[abx]);
    g2 += - g2C1a[abx] + g2C1b[abx] + g2C1c[abx]
          - g2C1e[abx] + g2C1f[abx];
    If[x < 0, g2 *= -1]; (* Since g2(x,q) = -g2(-x,-q) *)
    g2star = Sqrt[1-x^2]g2;
  ,((abq == 1) && (abx < 1)),
    (* Compute g2(abx, ± 1) *)
    If[((q == 1) && (x ≥ 0)) || ((q == -1) && (x < 0)),
      If[outputFlag == 3, Print["Case 3.2: x ∈ [0,1), q = 1
         OR x ∈ (-1,0], q = -1"]];
      g2 = 2/(1+3*abx)ArcSin[(1+abx)/2]Log[(3+abx)/2]
           -(Pi*Log[2])/(2(1+abx));
      g2 += (4*Log[Abs[(1-abx)/2]])/(Sqrt[3+abx] Sqrt[1-abx]);
      g2 += 1/(1-abx)*(-Pi + 4*ArcSin[(1+abx)/2]);
      g2 += Log[Abs[1-abx]]/Sqrt[1-abx^2](Log[Abs[1-abx]]
            -Log[1+3*abx] + 2*abx*g2C2dstar[abx]);
      g2 += g2C2a[abx] + g2C2b[abx] + g2C2c[abx] - g2C2e[abx]
           - 2*g2C2f[abx] - 2*g2C2g[abx];
      If[x < 0, g2 *= -1]; (* g2(-abx,-1) = -g2(abx,1) *)
      g2star = Sqrt[1-x^2]g2;
    , (* else *)
      If[outputFlag == 3, Print["Case 3.3: x ∈ [0,1), q = -1,
        OR x ∈ (-1,0), q = 1"]];
      g2 = (Pi*Log[2])/(2(1-abx))
           - 2/(1-abx)ArcSin[(1+abx)/2]Log[(3+abx)/2];
      g2 += -(4*Sqrt[1-abx]*Log[Abs[(1-abx)/2]])/(Sqrt[3+abx]
```

```
        *(1+3*abx));
      g2 += -4/(1+3*abx)*ArcSin[(1+abx)/2] + Pi/(1+abx);
      g2 += Log[1+abx]/Sqrt[1-abx^2]*(Log[Abs[1-abx]]
        -Log[1+3*abx] + 2*abx*g2C2dstar[abx]);
      g2 += -g2C3a[abx] + g2C3b[abx] + g2C3c[abx] - g2C3e[abx]
        + 2*g2C3f[abx] - 2*g2C3g[abx];
      If[x < 0, g2 *= -1]; (* g2(-abx,1) = -g2(abx,-1) *)
      g2star = Sqrt[1-x^2]g2;
    ];
  ,((x == 0) && (abq < 1)),
    If[outputFlag == 3,
      Print["Case 3.4: x = 0, q ∈ (-1,1)\{0}"]];
    (* Compute g2 (0, abq) *)
    g2 = 1/abq*ArcSin[abq/2](2*Log[(3*abq)/2]-Log[abq/2])
      -Pi/2*(Log[1+abq]-Log[Abs[1-abq]]);
    g2 += (1-abq)/(abq*(1+abq))*ArcSin[(1+abq)/2]
      *Log[Abs[(1-abq)/2]];
    g2 += -ArcSin[abq]/abq*(Log[1-abq] - Log[abq]);
    g2 += -g2C4a[abq] + g2C4b[abq] + g2C4c[abq]
      +1/abq*(g2C4d[abq] - g2C4e[abq]);
    g2 += -g2C4f[abq] + g2C4g[abq];
    If[q < 0, g2 *= -1]; (* g2(0, -abq) = -g2(0, abq) *)
    g2star=Sqrt[1-x^2]g2;
  ,True, (* else *)
    If[outputFlag == 3, Print["Case 3.5: x = 1, q ∈ [-1,1)
      OR x = -1,  q ∈ (-1,1]"]];
    If[x == 1, g2star = -2Log[Abs[1-q]],
```

```
        g2star = 2Log[Abs[1+q]]];
    ];
    If[outputFlag == 3, Print["g2star = ", N[g2star]]];
    T3 = c2 (-Sqrt[1-x^2]g1 + (q-x)*g2star);
  ];
  If[outputFlag == 3, Print["T3 = ", N[T3]]];

  (* Finally, combine terms T1-T3 to obtain the value of the
     kernel. *)
  If[outputFlag ≠ 0, Print["T1 = ", T1, ",  T2 = ", T2, ",
    T3 = ", T3, ",  k(x,q) = ", N[T1 + T2 - T3]]];
  N[T1 + T2 - T3]
]

kernel::usage = "kernel[x,q, nu, gamma0] gives the value of the
kernel k(x,q) where (x,q) ∈ S = {(x,q) ∈ [-1,1]^2 : x ≠ q} and
returns zero if (x,q) ∈ D = {(x,q) ∈ [-1,1]^2 : x = q}"
```

Listing D.4 `kernel_funcs.m`

```
(* ************************************************************
File: kernel_funcs.m
Author: Lauren Ferguson
Created: July 2010
Last Updated: June 2012

This package defines a number of functions needed by the routine
kernel[x,q] in the package "kernel.m", which is in turn called
from one of the routines in the package "ComputeL2Norm.m". Each
of the functions defined below is an integral that is computed
via numerical integration. The integrands are free from any
singularites over the interval of integration, as long as the
values of x and q correspond to the expected range of values
give for a particular case. These expected values are stated
in the comments given at the beginning of each case, however,
it is up to the user to ensure that the values sent to these
functions are indeed in the appropriate range, since we will
not check this condition in general.

First, we define the global precision (i.e., maxRecursion) used
for the numerical integration. However, on some integrals, we
will increase the precision to get a better result. Then, we
go through all the cases requiring these numerically integrated
functions. We list the accepted values of (x,q) in the comments
and define the functions needed for each case.
************************************************************ *)
```

```
maxRecursion := 12;


(* Term 1 functions: *)
(* Case 5: (x,q) ∈ (-1,1)^2, x < q *)
I1C6[x_,q_]:=If[x ≥ q,
                  Print["Error: I1C6 requires x < q"];
                  Abort[];
                , (* else *)
                  NIntegrate[ArcSin[r]/((r-x)^2),
                    {r,q,1}, MaxRecursion->maxRecursion]]

(* Case 6: (x,q) ∈ (-1,1)^2, x > q *)
f1a[x_,q_]:=NIntegrate[1/(Sqrt[1+r](Sqrt[2] + Sqrt[1+r])
               *(Sqrt[1-r] Sqrt[1+x] + Sqrt[1-x] Sqrt[1+r])),
               {r,q,1}, MaxRecursion->maxRecursion]
f1b[x_,q_]:=NIntegrate[(r(x-1)-(x+3))/(Sqrt[1+r](Sqrt[2]
            + Sqrt[1+r])(Sqrt[1-r](1+x) + Sqrt[1-x](1+r))),
             {r,q,1}, MaxRecursion->maxRecursion]


(* Term 3 functions: *)
(* Case 1: x ∈ (0,1), q = 0 *)
g2C1a[x_]:=NIntegrate[ArcSin[r]/(r(r-x)),
               {r,-1,0,x/2}, MaxRecursion->maxRecursion]
g2C1b[x_]:=NIntegrate[(ArcSin[r]*Log[Abs[r]])/((r-x)^2),
               {r,-1,0,x/2}, MaxRecursion->15]
g2C1c[x_]:=NIntegrate[(Log[Abs[r]]-Log[Abs[x]])
```

```
                /(Sqrt[1-r^2](r-x)),
                {r,x/2,x,(1+x)/2}, MaxRecursion->13]
g2C1d[x_]:=NIntegrate[(r+x)/(Sqrt[1-r^2]
                (Sqrt[1-x^2]+Sqrt[1-r^2])),
                {r,x/2,(1+x)/2}, MaxRecursion->maxRecursion]
g2C1e[x_]:=NIntegrate[ArcSin[r]/(r(r-x)),
                {r,(1+x)/2,1}, MaxRecursion->maxRecursion]
g2C1f[x_]:=NIntegrate[(ArcSin[r]*Log[Abs[r]])/((r-x)^2),
                {r,(1+x)/2,1}, MaxRecursion->maxRecursion]

(* Case 2: x ∈ [0,1), q = 1 *)
g2C2a[x_]:=NIntegrate[ArcSin[r]/((1-r)(r-x)),
                {r,-1,-((1+x)/2)}, MaxRecursion->maxRecursion]
g2C2b[x_]:=NIntegrate[(ArcSin[r]*Log[Abs[1-r]])/((r-x)^2),
                {r,-1,-((1+x)/2)}, MaxRecursion->maxRecursion]
g2C2c[x_]:=NIntegrate[(Log[Abs[1-r]]-Log[Abs[1-x]])
                /(Sqrt[1-r^2](r-x)),
                {r,-((1+x)/2),x,(1+x)/2}, MaxRecursion->maxRecursion]
(* Note: we reuse the next function in Case 3, since it is
   independent of q. *)
g2C2dstar[x_]:=NIntegrate[1/(Sqrt[1-r^2](Sqrt[1-x^2]
                    + Sqrt[1-r^2])),
                    {r,0,(1+x)/2}, MaxRecursion->13]
g2C2e[x_]:=NIntegrate[(Sqrt[1-r]Log[Abs[1-r]])/((1+r)^(3/2)(r-x)),
                {r,(1+x)/2,1}, MaxRecursion->maxRecursion]
g2C2f[x_]:=NIntegrate[(Sqrt[1-r]Log[Abs[1-r]])/(Sqrt[1+r](r-x)^2),
                {r,(1+x)/2,1}, MaxRecursion->maxRecursion]
```

```
g2C2g[x_]:=NIntegrate[ArcSin[r]/(r-x)^2,
                {r,(1+x)/2,1}, MaxRecursion->maxRecursion]


(* Case 3: x ∈ [0,1), q = -1 *)
g2C3a[x_]:=NIntegrate[ArcSin[r]/((1+r)(r-x)),
                {r,(1+x)/2,1}, MaxRecursion->maxRecursion]
g2C3b[x_]:=NIntegrate[(ArcSin[r]*Log[Abs[1+r]])/((r-x)^2),
                {r,(1+x)/2,1}, MaxRecursion->maxRecursion]
g2C3c[x_]:=NIntegrate[(Log[Abs[1+r]]-Log[Abs[1+x]])
                /(Sqrt[1-r^2](r-x)),
                {r,-((1+x)/2),x,(1+x)/2}, MaxRecursion->maxRecursion]
(* g2C3dstar[x_]:= g2C2dstar[x_] as above. *)
g2C3e[x_]:=NIntegrate[(Sqrt[1+r]Log[Abs[1+r]])/((1-r)^(3/2)(r-x)),
                {r,-1,-((1+x)/2)}, MaxRecursion->maxRecursion]
g2C3f[x_]:=NIntegrate[(Sqrt[1+r]Log[Abs[1+r]])/(Sqrt[1-r](r-x)^2),
                {r,-1,-((1+x)/2)}, MaxRecursion->maxRecursion]
g2C3g[x_]:=NIntegrate[ArcSin[r]/((r-x)^2),
                {r,-1,-((1+x)/2)}, MaxRecursion->maxRecursion]


(* Case 4: x = 0, q ∈ (0,1) *)
g2C4a[q_]:=NIntegrate[ArcSin[r]/(r(r-q)),
                {r,-1,-(q/2)}, MaxRecursion->maxRecursion]
g2C4b[q_]:=NIntegrate[(ArcSin[r]*Log[Abs[q-r]])/(r^2),
                {r,-1,-(q/2)}, MaxRecursion->maxRecursion]
g2C4c[q_]:=NIntegrate[(Log[Abs[q-r]]-Log[Abs[q]])/(r*Sqrt[1-r^2]),
                {r,-(q/2),0,q/2}, MaxRecursion->maxRecursion]
g2C4d[q_]:=NIntegrate[((q-r)Log[Abs[q-r]])/(r*Sqrt[1-r^2]),
```

```
                {r,q/2,q,(1+q)/2}, MaxRecursion->maxRecursion]
g2C4e[q_]:=NIntegrate[(ArcSin[r]-ArcSin[q])/(r-q),
                {r,q/2,q,(1+q)/2}, MaxRecursion->maxRecursion]
g2C4f[q_]:=NIntegrate[ArcSin[r]/(r(r-q)),
                {r,(1+q)/2,1}, MaxRecursion->maxRecursion]
g2C4g[q_]:=NIntegrate[(ArcSin[r]*Log[Abs[q-r]])/(r^2),
                {r,(1+q)/2,1}, MaxRecursion->maxRecursion]
```

Listing D.5 kernel_test.nb

```
(* ************************************************************
File: kernel_test.nb
Author: Lauren Ferguson
Created: June 2012
Last updated: June 2012

This notebook tests the computation of the kernel to make sure
that all the cases in the kernel are being handled correctly.
Basically, we just take points in the ranges for different cases
and make sure the correct case is called and the correct value is
given.

Inputs:
    nu = Poisson's ratio
    gamma0 = first (nondimensionl) surface tension paramter
    outputFlag = a flag used to display output in the kernel
                 file that is not normally shown.

NOTE: you MUST first comment out the outputFlag line in kernel.m
for this program to work as intended.
************************************************************ *)

(* Inputs: *)
nu = .2315;
gamma0 = 0.001;
```

```
(* Again, we must first load the kernel package and the library
   of functions it requires. *)
SetDirectory[NotebookDirectory[]];
Get["kernel_funcs.m"]
Get["kernel.m"]

(* Next we call the function for various values of x and q. As
always, we assume x ≠ q. *)

(* ************************ Term 1 ************************** *)
outputFlag = 1;

(* Case 1.1: x ∈ [-1,1], q = 1 *)
kernel[.95, 1, nu, gamma0];

(* Case 1.2: x ≠ 1, q = -1 *)
kernel[.5, -1, nu, gamma0];

(* Case 1.3: x = -1 *)
kernel[-1, .5, nu, gamma0];

(* Case 1.4: x = 1 *)
kernel[1, -.5, nu, gamma0]; -2*c1

(* Case 1.5: x < q ∈ (-1,1) *)
kernel[.25, .5, nu, gamma0];
```

```
(* Case 1.6: x > q ∈ (-1,1) *)
kernel[0, -.25, nu, gamma0];

(* Diagonal Cases *)
(* Case 1.7: x = q = -1 *)
kernel[-1, -1, nu, gamma0];

(* Case 1.8: x = q ∈ (-1,1) *)
kernel[.25, .25, nu, gamma0];

(* ********************** Term 2 ********************** *)
outputFlag = 2;

(* Case 2.1: x = -1 *)
kernel[-1, .25, nu, gamma0]; 2*c2*(1 + .25)

(* Case 2.2: x = 1 *)
kernel[1, .25, nu, gamma0]; 2*c2*(1 - .25)

(* Case 2.3: x ∈ (-1,1) *)
kernel[.5, -1, nu, gamma0]; -c2*Pi*Sqrt[1 - (.5)^2]

(* Diagonal Case:  Case 2.4: x = q ∈ [-1,1] *)
kernel[.25, .25, nu, gamma0];

(* ********************** Term 1 ********************** *)
(* ********************** g2(x,q) ********************** *)
```

```
outputFlag = 3;

(* Case 3.1: x ∈ (0,1), q = 0; also covers x ∈ (-1,0)U(0,1),
   q ∈ (-1,1), x ≠ q *)
kernel[.5, .25, nu, gamma0]; kernel[-.5, -.25, nu, gamma0];

(* Case 3.2: x ∈ [0,1), q = 1 OR x ∈ (-1,0], q = -1 *)
kernel[.5, 1, nu, gamma0]; kernel[-.5, -1, nu, gamma0];

(* Case 3.3: x ∈ [0,1), q = -1 OR x ∈ (-1,0], q = 1 *)
kernel[.5, -1, nu, gamma0]; kernel[-.5, 1, nu, gamma0];

(* Case 3.4: x = 0, q ∈ (-1,0)U(0,1) *)
kernel[0, -.5, nu, gamma0]; kernel[0, .5, nu, gamma0];

(* Case 3.5: x = 1, q ∈ [-1,1) OR  x = -1, q ∈ (-1,1] *)
kernel[1, .25, nu, gamma0]; kernel[-1, -.25, nu, gamma0];

(* Diagonal Case:  x = q *)
kernel[1, 1, nu, gamma0];

(* ******************** Recheck Diagonal ********************** *)
outputFlag = 4;

kernel[-1, -1, nu, gamma0];
kernel[-.25, -.25, nu, gamma0];
kernel[0, 0, nu, gamma0];
```

```
kernel[.5, .5, nu, gamma0];
kernel[1, 1, nu, gamma0];
```